Linux 嵌入式实时操作系统开发与应用

By 杨立峰

感　谢

感谢我的父母，因为你们教育了我

Abstract

In the fast-changing world of science and technology. Appeared information appliances, handheld and wireless devices. There are many hardware and software design changes taking place. Many devices now feature 32-bit microprocessors from Intel, MIPS and Motorola, as well as larger LCD graphical displays. In order to leverage the significant results gained in the last ten years, many developers are turning to using friendly user interface operating systems with these new embedded designs.

One of the most promising emerging areas seems to be running Linux in these environments, for a couple of good reasons: Linux on embedded systems brings with it the entire power of desktop computing, along with many solutions already running. Linux, being open source, allows any aspect of the solution to be fully understood and then customized for a particular application. Linux also supports all the new microprocessors typically included in embedded designs, including StrongARM, MIPS and PowerPC. Finally, Linux is free, with no royalty payments required for its use. So using Linux as operating system , with a GUI system built on, seems to be a good solution.

For the handled devices on the market such as PDA, as to the poor hardware in old days, the function was very simple; we could hardly see the GUI and network support. But recently we found that some embedded operating systems such as Windows CE and Palm OS, have supported complete GUI features. With the great performance improvement of the hardware, we think that the need for embedded OS is urgent.

I got interest with Linux operating system several years ago. Combination my specialty, Then I did some research for real-time Linux. Based on these facts, this thesis demonstrates architecture and internals of Linux system used on embedded systems.

At first, the thesis outlines the history of embedded systems and real-time systems. Chapter 2 describes related research in area of real-time Linux systems. Chapter 3 details the design and implementation of real-time Linux. Chapter 4 contain a discussion of the application model of real-time Linux. The measurements results of real-time Linux performance can be found in Chapter 5. The last chapter gives some conclusion and foresight.

Keywords: Real Time System; Embedded System; RTLinux; Linux

摘　　要

伴随着科技不断的日新月异推陈出新。信息家电，手持设备，无线设备等个性化设备的出现，相应的硬件和软件的迅速发展。许多设备都配有Intel，MIPS，摩托罗拉等公司生产的 32 位微处理器。许多开发商也开始为这些设备提供嵌入式操作系统。嵌入式系统与实时系统将会有越来越重要作用。

利用 Linux 搭建嵌入式操作系统是近年来出现的最令人振奋的方案。这有多方面的原因。首先，运行在嵌入式系统上的 Linux 能够提供全功能的桌面计算（Desktop Computing），且由于其开放代码，定制变得非常方便。其次，Linux 已经支持大多数嵌入式系统上使用的芯片，包括StrongARM，MIPS 和 PowerPC。最后，Linux 是免费的，使用 Linux 不需要付出任何费用。现今风行世界的 Linux 操作系统，本着开放自由的精神，吸引了全世界的目光，越来越多的程序员加入到它的行列中来。所以利用 Linux 作为底层操作系统, 在其上进行实时化改造，建立一个具有实时应用能力的操作系统是现在日益流行的嵌入式操作系统的解决方案。

市场上常见的 PDA 等小型手持式设备上，以前由于硬件条件等的限制，我们看到功能都非常简单，没有网络、GUI 等非常实用方便的功能。但最近出现的 Palm 等手持式电脑或者在 Windows CE 等面向嵌入式系统的操作系统上，我们已经看到了完整的网络、图形用户界面支持。随着手持式设备的硬件条件的提高，估计嵌入式系统对嵌入式操作系统的需求会越来越迫切。

本人长期以来对 Linux 操作系统比较感兴趣，并结合本专业，对 Linux 应用于嵌入式实时环境进行了一定的研究。本论文基于这些事实，对面向嵌入式实时环境的 Linux 系统的体系结构和一些技术内幕做了较深入的探讨。

论文首先概述了嵌入式系统及实时系统的发展情况。第二章介绍了在实时 Linux 领域的相关研究。第三章介绍了典型的实时系统 RTLinux 的设计与实现，第四章介绍了 RTLinux 的编程模型和一些应用实例。本文的第五章将对 RTLinux 的性能进行测试评估。最后是一点感想和展望。

关键词：　Linux；　RTLinux；　实时系统；　嵌入式系统。

目 录

图表及程序目录

缩略语

API Application Program Interface

ATM Automated Teller Machine

CPU Central Processor Unit

DMA Direct Memory Access

EDF Eariest Deadline First

FIFO First-In-First-Out

GUI Graphical User Interface

I/O Input/Output

IPC Interprocess Communication

ISR Interrupt Service Routine

IST Interrupt Service Thread

MCU Micro-Controller Unit

OS Operating System

PC Personal Computer

PDA Personal Data Assistant

POS Point Of Sells

POSIX Portable Operating System Interface for computer Environments

RAM Random-Access Memory

ROM Read-Only Memory

RT Real Time

RTOS Real Time Operating Systems

TLB Translation Lookaside Buffer

第一章　嵌入式实时系统概况

1.1　嵌入式系统概况

1.1.1　嵌入式技术的历史发展

嵌入式系统的出现至今已经有 30 多年的历史了，嵌入式技术也历经了几个发展阶段。进入 90 年代后，以计算机和软件为核心的数字化技术取得了迅猛发展，不仅广泛渗透到社会经济、军事、交通、通信等相关行业，而且深入到家电、娱乐、艺术、社会文化等各个领域，掀起了一场数字化技术革命。多媒体技术与 Internet 的应用迅速普及，消费电子(Consumptive electron),计算机(Computer),通信(Communication),3C 一体化趋势日趋明显，嵌入式技术再度成为一个研究热点。综观嵌入式技术的发展，大致经历了以下 4 个阶段[1]。

- 第一阶段是以单芯片为核心的可编程控制器形式的系统，同时具有与监测、伺服、指示设备相配合的功能。这种系统大部分应用于一些专业性极强的工业控制系统中，一般没有操作系统的支持，通过汇编语言编程对系统进行直接控制，运行结束后清除内存。这一阶段系统的主要特点是：系统结构和功能都相对单一，处理效率较低，存储容量较小，几乎没有用户接口。由于这种嵌入式系统使用简便、价格很低，以前在国内工业领域应用较为普遍，但是已经远远不能适应高效的、需要大容量存储介质的现代化工业控制和新兴的信息家电等领域的需求。
- 第二阶段是以嵌入式 CPU 为基础、以简单操作系统为核心的嵌入式系统。这一阶段系统的主要特点是：CPU 种类繁多，通用性比较弱；系统开销小，效率高；操作系统具有一定的兼容性和扩展性；应用软件较专业，用户界面不够友好；系统主要用来控制系统负载以及监控应用程序运行。
- 第三阶段是以嵌入式操作系统为标志的嵌入式系统。这一阶段系统的主要特点是：嵌入式操作系统能运行于各种不同类型的微处理器上，兼容性好；操作系统内核精小、效率高，并且具有高度的模块化和扩展性；具备文件和目录管理、设备支持、多任务、网络支持、图形窗口以及用户界面等功能；具有大量的应用程序接口（API），开发应

用程序简单；嵌入式应用软件丰富。

- 第四阶段是以基于 Internet 为标志的嵌入式系统，这是一个正在迅速发展的阶段。目前大多数嵌入式系统还孤立于 Internet 之外，但随着 Internet 的发展以及 Internet 技术与信息家电、工业控制技术等结合日益密切，嵌入式设备与 Internet 的结合将代表着嵌入式技术的真正未来。

1.1.2 嵌入式系统的技术特点和应用前景

嵌入式系统主要由嵌入式处理器、相关支撑硬件和嵌入式软件系统组成，它是集软硬件于一体的可独立工作的“器件”。嵌入式处理器主要由一个单片机或微控制器(MCU)组成。而这些嵌入式 CPU 目前多是 8 位和 16 位的，与 32 位或 64 位的高性能处理器相比，具有很强的经济性和现实性。相关支撑硬件包括显示卡、存储介质（ROM 和 RAM 等）、通讯设备、IC 卡或信用卡的读取设备等。嵌入式系统有别于一般的计算机处理系统，它不具备像硬盘那样大容量的存储介质，而大多使用闪存（Flash Memory）作为存储介质。嵌入式软件包括与硬件相关的底层软件、操作系统、图形界面、通讯协议、数据库系统、标准化浏览器和应用软件等。

总体看来，嵌入式系统具有便利灵活、性能价格比高、嵌入性强等特点，可以嵌入到现有任何信息家电和工业控制系统中。从软件角度来看，嵌入式系统具有不可修改性、系统所需配置要求较低、系统专业性和实时性较强等特点。

后 PC 时代是一个真实的阶段，而且是一个可以预测的时代。嵌入式系统就是与这一时代紧密相关的产物，它将拉近人与计算机的距离，形成一个人机和谐的工作与生活环境。从某一个角度来看，嵌入式系统可应用于人类工作与生活的各个领域，具有极其广阔的应用前景。嵌入式系统在传统的工业控制和商业管理领域已经具有广泛的应用空间，如智能工控设备、POS/ATM 机、IC 卡等；在家庭领域更具有广泛的应用潜力，如机顶盒、数字电视、WebTV、网络冰箱、网络空调等众多消费类和医疗保健类电子设备等；此外还有在媒体手机、袖珍电脑、掌上电脑、车载导航器等方面应用，将极大地推动嵌入式技术深入到生活和工作的方方面面。它在娱乐、军事方面的应用潜力也是巨大的，而且是有目共睹的。

1.1.3 典型的嵌入式系统

信息家电商机引发全球嵌入式操作系统平台大战，全球 4 大操作系统阵营 WinCE[18]、Palm OS[19]、EPOC[20]和 Linux[21]展开规格战，各拥有软件及硬件合作厂商逐鹿信息家电市场的份额。

全球手持式信息家电快速增长，据预测，2000 年至 2004 年市场增长率将到达 77.4%，个人数字助理器（PDA）、智慧型手机等手持式信息家电操作系统竞争日益激烈。除了为后个人电脑时代的硬件大厂带来新一轮商机外，应用软件厂商包括电子字典、电子地图、游戏开发业者的商机也大量涌现。

微软窗口操作系统拥有在个人电脑上的操作系统占有率的优势，使 WinCE 拥有强大的窗口资源支援。不过 Palm OS 操作系统拥有全球 PDA 产品 70%的市场占有率；同时获得 3COM、IBM 和索尼等跨国公司的支持。EPOC 是发展自欧洲的操作系统、是由世界上最大的 3 家移动电话厂商诺基亚、爱立信和摩托罗拉所共同开发、整合组成新公司，开发出来的新操作系统；在 3 大电话厂商的合作下，EPOC 市场潜力很大，且占有率高，但应用功能以手机为主，目前并不开放授权。

此外，在 3 大主流操作系统品牌外，Linux 也将是今后一股强劲的力量；由于 Linux 开放源码，经过这些年的发展，已经成为一个健壮的可靠的高性能的操作系统。愈来愈多的嵌入式系统设计员发现 Linux 可以成为一个优秀的嵌入式操作系统。而 Linux 的最大的优势还在于它是一个开放的操作系统。由于 Linux 开放源码，操作系统的一切对用户都是透明的，用户可以最大限度地控制系统开发的进度和造价。在开发过程中遇到的各种各样的硬件设备，可以方便地在网上找到这些设备的驱动程序，得到支持。Linux 内置网络支持，用户可以轻松地使自己的嵌入式具有网络功能。Linux 是模块化的操作系统，提供了优秀的可缩放功能，用户可以方便地删除不需要的模块，大多数嵌入式系统对操作系统的体积非常敏感，Linux 的可以根据自己的需要，选择特定的功能模块，自主地搭建嵌入式操作系统。Linux 支持绝大多数 CPU，包括 Intel、MIPS、ASIC、ALPHA、68K、POWER PC 等。这使 Linux 几乎可以嵌入到各种硬件设备上。成为各家厂商极力发展的操作系统，加上其核心小，潜力可观。

1.2 实时系统概况

1.2.1 什么是实时系统

实时计算正在成为越来越重要的原则。操作系统，特别是调度程序，可能是实时系统中最重要的组件。实时系统的例子包括实验控制、过程控制设备、机器人、空中交通管制、远程通信、军事指挥与控制系统，下一代系统还将包括自动驾驶汽车、具有弹性关节的机器人控制器、智能化生产中的系统控制、空间站和海底勘探等。

实时计算[5]可以定义成这样一类计算，即系统的正确性不仅取决于计算的逻辑结果，而且还依赖于产生结果的时间。我们可以通过定义实时进程或实时任务来定义实时系统。一般说来，在实时系统中，某些任务是实时任务，它们具有一定的紧急程度。这类任务试图控制外部世界发生的事件，或者对事件做出反应。由于这些事件是“实时”发生的，因而实时任务必须能够跟得上它所关心的事件。因此，通常给一个特定的任务制定一个最后期限，最后期限指定开始时间或结束时间。这类任务可以分为硬任务或软任务两类。一个硬实时任务（hard real-time task）指必须满足最后期限的限制，否则会给系统带来不希望的破坏或者致命的错误。一个软实时任务（soft real-time task）也是一个与之关联的最后期限，并希望能满足这个期限的要求，但这并不是强制的，即使超过了最后期限，调度和完成这个任务仍然是有意义的。实时任务的另一个特征是它们是周期还是非周期的。一个非周期任务（aperiodic task）有个必须结束或开始的最后期限，或者有一个关于开始时间和结束时间的约束。而对于周期任务（periodic task），这个要求描述成“每隔周期 T 一次”或者“每隔 T 个单位”。

1.2.2 实时操作系统的特点

实时操作系统一般符合以下的一些要求[5]：

- 可确定性：是指它可以按照固定的、预先确定的时间或时间间隔执行操作。
- 响应性：是指在系统得到中断后系统为中断提供服务的时间。
- 用户控制：是指允许用户细粒度地控制任务的各种属性（任务优先级、任务权限等）。
- 可靠性

- 故障弱化运行：是指系统在故障时尽可能多地保存其权能和数据的能力。

嵌入式系统一般都是实时系统，实时系统大都是用在嵌入式环境。用于嵌入式场合的实时操作系统就是嵌入式实时操作系统。在一个较为完善的嵌入式系统环境中，需要一个支持实时多任务的操作系统(RTOS)内核，因为广泛应用于现实世界的嵌入式设备必须具有与外部环境实时交互的能力。RTOS 是和嵌入式应用复杂化直接相关的，在应用需求的复杂度不断增加的今天，如果实时应用软件开发还是没有基于一个完善的 RTOS，那么无法将系统软件和应用软件分离，开发周期过长、成本过高。因此，因此 RTOS 是实时应用软件开发的必然产物。

第二章　实时系统的相关研究

尽管现在的操作系统变得种类繁多，但是 UNIX 及其兼容的系统仍然是工业和学术领域标准的操作系统。一些非 UNIX 系统，比如 Windows NT，也是与 POSIX.1003 标准兼容，这个标准无疑是基于 UNIX。这个系统的成功是由于它的开放性、稳定性和事实的标准。随着 POSIX1003.1b 实时扩展标准的发布，UNIX 有机会成为分布最广泛的实时处理平台。Linux 作为一个类 UNIX 的系统，凭借其开放源码的优势，获得越来越广泛的应用。

由于以上的原因，在这一章我将集中讨论与 Linux 相关的实时系统。我将讨论在 Linux 上进行实时运算的问题，和在一些系统中如何解决这些问题。

2.1　LINUX 的分时特性

UNIX 最初是作为一个分时系统设计的[17]。LINUX 作为 UNIX 的克隆，很多当前的实现中仍然保留了这些特点。它们力争最优的平均性能。这个目标通常与实时系统的低延迟和高可预言的要求相勃的。为了说明这个问题，让我们考虑一个通过扬声器发声的程序（程序 2.1）。

```
#define DELAY 10000
main()
{
     int i;
     while (1) {
         for (i=0;i<DELAY;i++)
             speaker_on();
         for (i=0;i<DELAY;i++)
             speaker_off();
     }
}
```

程序 2.1　简单的发声程序

扬声器的驱动程序假定为只有两种状态 on 或 off。初看起来这个程序可以按给定的周期输出方波，使扬声器正常的发声。然而，当运行一个标准的 LINUX 程序时，它将不能正确的发声。

我在一个有 412MHz Celeron 处理器的 Linux 操作系统上运行这个程

序。当在系统没有别的程序在运行时，扬声器发出稳定的声音。每一个滴答声都可以听到。当有按键动作或者移动鼠标时，都会引起声音的断续。在执行磁盘操作或高运算量的程序时，声音将变得严重失真。最后，起动一个大的程序，比如 X-Windows，扬声器将持续大约半秒的时间不能发声。假如这个程序是控制步进电机的，而不是使扬声器发声，那么程序将不能使电机稳定地运行。

Linux 的设计和实现的原理大体上与 UNIX 是相同的[12]。它们都是采用分时的调度，低的计时分辨率，非占先式内核，关中断和虚拟内存。我们在细节上来考虑这些问题。

调度程序是内建在操作系统内部的一组策略和机制，它决定哪一项工作将由计算机来完成[4]。

大部分的 UNIX 操作系统，尤其是 Linux 操作系统，它们的调度程序追求的是平均响应时间、吞吐量和在进程之间的公平的 CPU 时间分配[16]。每个进程的优先级是动态的基于进程已经花费的 CPU 时间，输入/输出强度和别的一些因素来决定。

Linux 系统使用固定的时间片(time slices)来调度 CPU 时间。最开始进程赋予一个高的优先级。如果在某个进程的时间片内，这个进程放弃 CPU，它的优先级将不会变，或者变的更高。另一方面，如果一个进程使用完它的时间片，它的优先级将会变低。这种策略关心的是交互式程序，比如说编辑器，由于这类程序更多的把时间花费在等待 I/O 输入输出的完成。虽然对在终端前的用户来说是有利的。由于程序的执行完全依赖于复杂的、不可预知的系统负荷与别的进程的活动，这种调度方式对于实时进程而言完全没有用。

Linux 中加进了 POSIX 实时扩展部分，引进了实时进程的概念，允许一个进程定义为一个实时进程。Linux 区分实时进程和普通进程，采用不同的调度策略。即先来先服务调度(SCHED_FIFO)和时间片轮转调度(SCHED_RR)。在 SCHED_RR 调度中，任务一旦时间片用完就被移动到优先级队列的队尾，并允许同一优先级的其它任务运行。如果同一优先级没有其它任务，该任务继续运行下一个时间片。SCHED_FIFO 是运行直至阻塞的策略。SCHED_FIFO 任务按优先级调度，一旦开始就一直运行到结束或阻塞在某种资源上。不像 SCHED_RR 任务那样共享处理器。

另外还有计时器的精度问题。以前提供给用户进程警报信号和 sleep() 系统调用只有 1 秒的精度，如此粗糙的计时精度是不适合大多数的实时进

程。当前的版本提供了更高精度的时间间隔，然而，内在的时钟实现限制了计时的正确性。这方面的内容在后面将由更详细的论述。

大部分的 Linux 的核心进程是不能中断的[10]。换句话说，一旦一个进程进入到核心模式，它将运行到系统调用的完成或者被阻塞为止。假如在这期间有一个更高优先级的实时进程准备好运行了，它将不得不等待。由于不需要考虑内核重入的问题，这种设计的方式使内核的开发更为简单。然而，一个系统调用可能花费很长时间来完成，对于一个实时进程来说长的延迟是不能接受的。

于非占先式内核相关的问题是系统的同步。为了保护数据可能被非同步的操作，比方说中断处理函数，系统设计者通常在临界区代码中选择关中断的方式来处理。比起信号量(semaphores)或者自旋锁(spinlocks)这是更为简单有效的技术。但是，禁止中断是系统能力与系统对外部事件的快速响应的一个折中。这种方法还是不能解决多处理器系统的同步化问题。

Linux 系统使用了虚拟内存用于分页[10]。虚拟内存技术只是保护程序在运行部分在 RAM 中，可以使运行的程序超过系统 RAM 的容量。这种方式在分时系统中将很好的运行。然而，对于实时系统来说，虚拟内存引起的系统不可确定性达到一个无法忍受的地步。

所有考虑的这些因素来看，显然传统的 Linux 是不可能用于实时处理。我们需要一些根本的改变。

2.2 Linux 性能测试

为了对 Linux 的性能有一个直观的了解，我对 Linux 系统进行了测试。测试的内容包括中断延迟时间和上下文切换。对测试的结果进行分析，以寻找提高 Linux 延时间性能的途径。

2.2.1 中断延迟测试

中断可以分为两种不同类型：同步和非同步中断。对应用程序来说，重要的是非同步中断。非同步中断发生的情况如图 2.1 所示。中断响应时间是中断发生到中断处理程序开始执行之间的时间差。这个时间差包括直到在运行任务停止和中断分派时间。

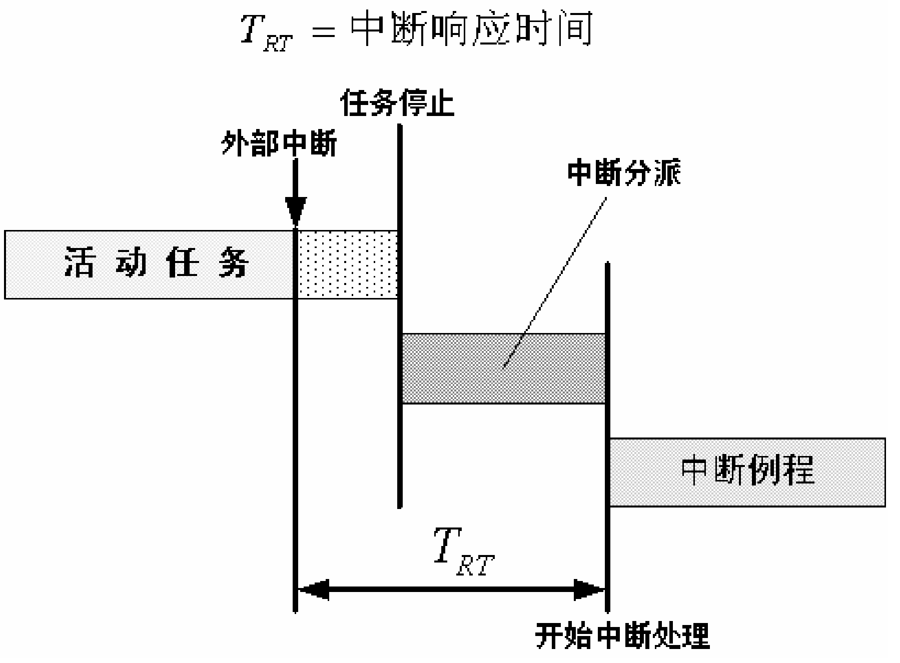


图 2.1　异步中断和中断响应时间

中断响应时间并不是一个常量。它与操作系统和硬件平台有关。要测量精确的关闭中断的时间，并不是通过上面的定义来进行。因为从中断到来到当前任务停止属于中断延迟时间。在 Linux 中，内核或驱动程序显式地关/开中断，一般是通过调用__cli()/__sti()来进行操作。中断延迟程序计算一对__cli()/__sti()调用之间的时间。在调用__cli()时，记录系统时间值，读出__sti()被调用时的系统时间值。他们之间的时间差就是关中断时间。Linux 下的关中断时间如图 2.2 所示：

关中断时间测试程序重新写了__cli()/__sti()宏，以允许记录调用它们的文件以及在何处调用。记录这些信息以分析在 Linux 中那些关中断时间是比较长。（中断测试程序的代码在附录 A）

我对 Linux 进行了大约 3 个小时的测试，测试的结果如表。在测试中运行一些程序，其中包括一个磁盘循环拷贝程序，打开一些应用程序。可以发现系统负载比较重时，系统的页面调度花了比较多的时间，将近 500 微秒。表 2.1 表 2.2 是统计结果。

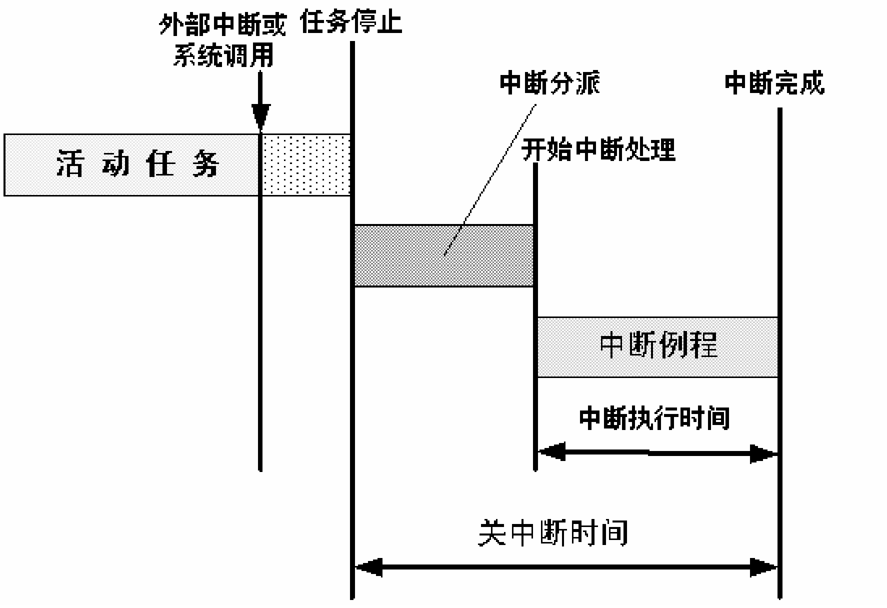


图 2.2　关中断时间

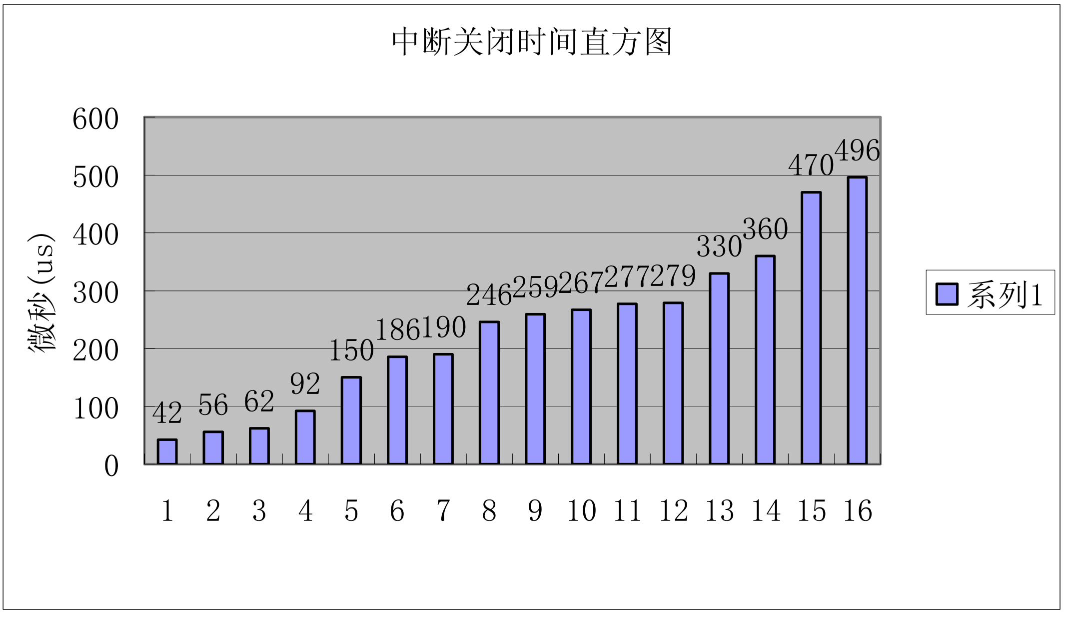


表 2.1　中断关闭时间直方图

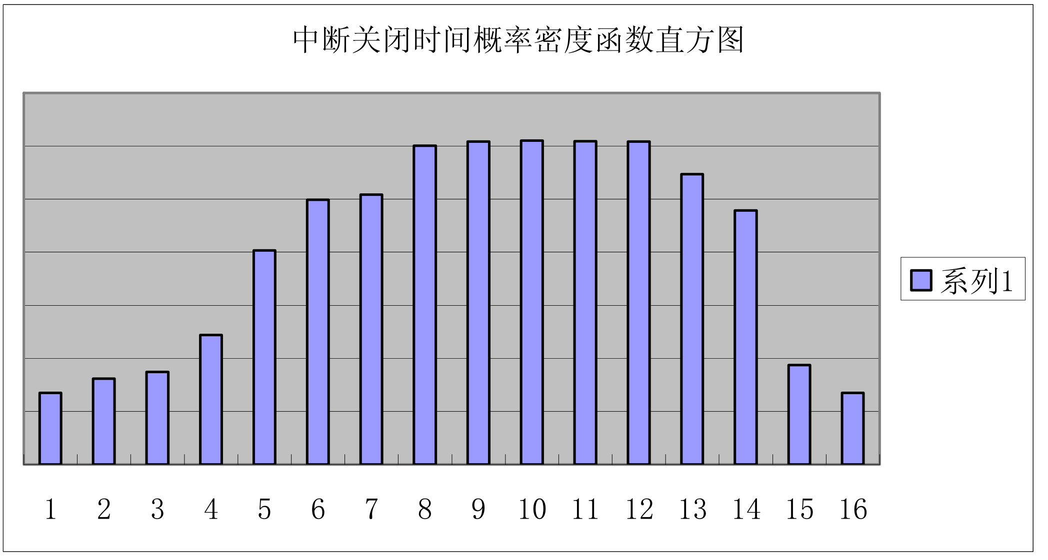


表 2.2　中断关闭时间概率密度函数直方图

可以看出在我的测试系统中系统中断关闭时间最大达到 496 微秒，一般中断关闭时间是在 250 微秒到 300 微秒左右。这次测试并没有进行所有情况下的测试，从这些结果我们就可以看出： Linux 的系统设计人员采用分时的调度、低的计时分辨率、非占先式内核、关中断和虚拟内存是造成系统关中断时间过于长的原因。

2.2.2　上下文切换测试

上下文切换时间是保存一个进程状态，然后恢复另外一个进程状态的时间。我写了一个测试程序来测试这个时间（程序见附录 B）。程序运行时，根据输入的参数来决定创建多少个进程。所有的进程用一个环形的 UNIX 管道连接。程序中实现一个令牌在这些进程之间传递，迫使进行进程间的上下文切换。程序记录在进程间传递令牌 2000 次所花的时间。每一次令牌的传递有两个开销：上下文切换开销和令牌传递开销。程序首先计算令牌在环形管道中传递的开销，在输出的结果已经除去了这部分开销。

为了计算更真实的切换时间，我加入了人为的数据在里面，进程切换时间包括保存用户级数据状态的时间。测试的结果在表 2.3 所示，Y 轴表示切换时间，X 轴表示进程数目，size 表示进程的大小。

从结果看，进程的随着进程的大小变化，切换时间在增加，在 16k 以内增加幅度不大，是因为此时进程的大小还没有超出一级缓存的大小，超过 16k 时，增加幅度比较大，进程大小达到 64k 时，切换时间达到 300 微秒。Linux 切换时间过于大的原因是系统保存了过于多的状态。在上下文切换过程中，系统是关中断的，意味着此时的系统关中断时间超过 300 微秒。对于实时应用来所是不能接受的。

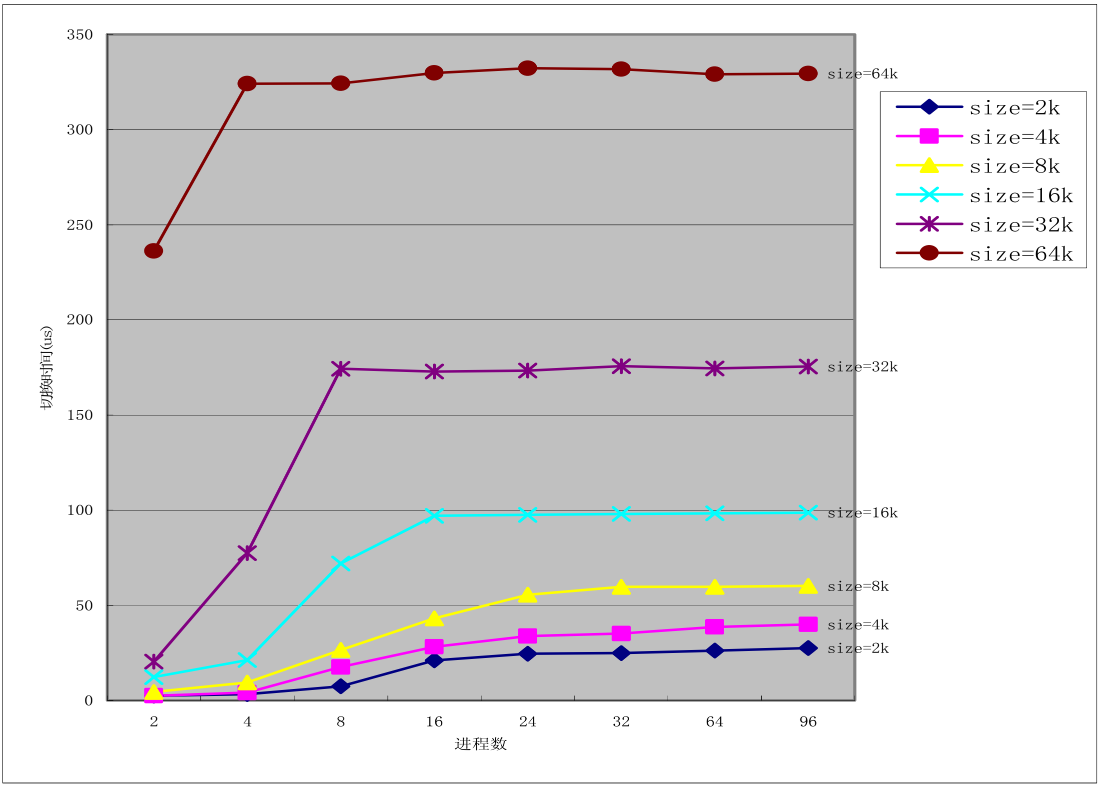


表 2.3　上下文切换时间

2.3　当前的实时操作系统

在这一节我们来看看一些操作系统的设计者是怎样来处理前一节所提到的问题的。

最简单的解决方案是改变分时的调度程序。一个例子是文献[12]介绍的系统。MINIX 的 round-robin 调度器换为基于优先级的调度器。由于在 MINIX 中不使用页面调度和页面交换技术，假如对时间的响应不过分要求

的话，这种方法是可以接受的。

一些在 UNIX 系统中采用 POSIX.1b-1993 实时标准规定的方式。这个标准规定了优先级调度，锁定用户程序页面，实时信号，改进的 IPC 和计时器，和别的一些特性。依从这些标准使 UNIX 更适合实时应用。

Linux 是部分支持 POSIX.1b 标准的操作系统[12]。从 1997 年 5 月 1 日起，在 Linux 中完全实现了适合用于控制的调度器和内存锁定技术，定时器也部分实现。内核的不可占先，低的定时精度，和高的中断延迟仍然没有解决。因此，POSIX.1b 兼容的 Linux 操作系统只适合一些软实时的处理。

别的满足 POSIX.1b 的操作系统是 QNX[26]。QNX 操作系统是微内核的操作系统。内核只是实现 4 种服务：进程调度，进程间通信，低级的网络通信，和中断分派。别的一些服务，比如说设备驱动程序和文件系统，实现为与用户进程协同操作的方式。这样系统内核非常小（大约 7KB 左右）而且非常快的。

QNX 兼容 POSIX 1003.1 标准（编程接口）和 POSIX 1003.2 标准（外壳和应用程序）。对于熟悉 UNIX 的开发者来说，是非常方便的。QNX 提供标准的 UNIX 部件：编译器，调试器，X-Windows，和 TCP/IP。

微内核的设计比起传统的单块结构的设计有很多优势。调试用户程序比起调试内核模块更为简单。假如用户进程运行在单独的地址空间（像 QNX 那样），内存管理错误在不同模块是相互隔离的。驱动程序可以更获得多线程的好处。另外有良好的裁剪性。比如，QNX 可以减少到 100KB 以下以适合 ROM 的大小，或者扩展到全功能的多机开发环境。移植和维护微内核的系统也更简单。缺点是微内核比起单块结构的内核来说不那么紧凑。

对于实时处理的微内核它提供了轻量级进程，快的上下文切换，和 IPC。一个实时的用户进程可以中断设备驱动程序，在单块结构的内核是不可能的。由于微内核操作系统是非常小的，可以方便的计算最坏情况下的计时参数，比如中断延迟时间。

大多数的微内核操作系统的弱点是它的性能要差。微内核结构的操作系统在进程间通信和上下文交换有比较重的系统开销。微内核操作系统只提供简单的系统服务。因此，与单块结构的操作系统相比，完成相同的任务微内核操作系统要进行更多的系统调用。虽然一些研究者认为上下文切换，消息传递等可以高效地实现[2]，就性能而言，单块结构的操作系统仍

然更成功。

一个单块结构的操作系统的例子是 VxWorks[27]。VxWorks 是专有的实时操作系统，采用主机/目标机方式。一个 UNIX 主机用来软件的开发和运行程序的非实时部分。一个叫 wind 的 VxWorks 内核在目标机上运行实时任务。机器间的通信使用 TCP/IP 网络连接。

虽然 VxWorks 不兼容 UNIX 系统，它提供了一些符合 POSIX 接口规范的函数，特别是 POSIX.1b 实时扩展部分。大多数的 VxWorks API 是专有的。

在 VxWorks 中，内核和任务运行在同一个地址空间。这样，任务间的切换非常快，和省去了必要的系统调用开销。一个实时连接器允许动态的装载任务和系统模块。这些特性使系统具有好的伸缩性。一个交互式的类似 C 语法的 shell 用来检查和改变参数值，计算表达式，调用函数，和进行简单的调试。这个特性使得开发更为简单方便。内核和任务运行在同一个地址空间也使得系统更脆弱，一个模块的错误更容易影响到别的模块。

REAL/IX 操作系统也是单块结构。这是一个有完全 UNIX 特性的操作系统，原自于 UNIX System V，改造使它拥有实时处理能力。它的内核是完全可占先的。它通过内核信号量来实现，提供互斥方式访问系统资源，比起传统的 sleep/wakeup 函数和关中断来说，这种方式更优秀。使用信号量代替关中断降低了中断延迟时间,也使的移植到多处理器系统更为容易。

REAL/IX 是 POSIX.1003 兼容的操作系统。这个特性使得移植 UNIX 应用程序更为容易。除实时调度以外，实时能力还包括预先分配内存和文件空间，同步与非同步 I/O，增强的 IPC 和计时器，可连接的中断。最后的特性允许用户进程处理中断。

现在有一个趋向于使用 Windows NT 来支持实时处理。只要原因是兼容先前的 Windows，因此，可以使用先前的应用程序。也是指望用一个操作系统来做任何事情：办公应用，服务，和实时控制。用一个操作系统可以减少人员培训的开销。程序员可以用到广泛的 Win32 API。Microsoft 对市场的垄断也是一个原因。

像文献[25]指出的那样，Windows NT 的内核并不适合硬实时的处理：Win32 API 不是设计用来实时应用的，中断处理的方式可能引起一个不确定的延迟，而且对于嵌入式系统来说，Windows NT 对内存的要求也是一个问题。

Microsoft 为了处理上面提到的一些问题。开发了 Windows CE 嵌入式

实时操作系统，Windows CE 内核支持按优先级抢占的方式调度多任务。Windows CE 可以固化到 ROM 中，从 ROM 起动，使用时对内存 RAM 要求不高。Windows CE 将中断分为两个步骤：第一步是执行中断服务子程序 ISR(interrupt service routine)；第二步为中断服务线程 IST(interrupt service thread)。内核使中断处理程序执行尽可能短，大部分处理放在 IST 中，可以把最高优先级安排给中断服务线程，以保证 IST 尽快运行而不会被占先。其与 Win32 API 兼容的编程接口方便熟悉 Windows 的程序员开发和移植 Windows 应用。

第三章　嵌入式实时 LINUX 系统 RTLINUX 的设计与实现

从上一章的分析，我们已经知道 Linux 是一个通用操作系统，将它应用于嵌入式实时环境有许多缺点和不足。特别是在运行内核线程时，Linux 关闭中断，别的问题包括分时的调度，虚拟文件系统的时间不确定性，缺乏高精度的计时器。所以要对现有的 Linux 进行改造，即要对 Linux 进行实时化，这一章将介绍 RTLinux 的结构和如何对 RTLinux 进行实时化。

3.1　RTLinux 的结构

RTLinux 使用众所周知的虚拟机技术的简单方案来解决这些相互对立的解决上面提到的问题。增加了一个仿真程序来替换 Linux 的底层中断程序。一个小的实时内核与 Linux 内核共享控制处理器。如果来自硬件的属于实时内核的中断将直接被处理，属于 Linux 内核的中断通过中断仿真程序处理。假如 Linux 内核中断请求没有被允许，中断模拟程序将在中断队列中标记这次中断的发生。当 Linux 内核的中断请求被允许时，在中断队列中的中断将被执行。因此，实时内核的操作可以得到机器的立即相应，而且 Linux 内核不能延迟实时任务的执行。实时任务与运行在 Linux 内核中的进程之间的通讯通过 FIFOs 与共享内存的方式进行。使用实时内核中的调度器调度实时任务，调度器的算法和策略可以用户自己定义；而系统也已经实现了 RMS 和 EDF 算法。

这样就保留了 Linux 操作系统所提供的丰富的功能，而且改动它使其作为一个基本内核与实时内核共享控制 CPU。实际上，系统可以看作具有双内核的操作系统，实时内核拥有更高优先级别的任务，换句话说，基本内核可以看作实时系统的空闲任务:只是在没有实时处理要求的时候运行。这样实现的 RTLinux 的详细结构图如图 3.1 所示。

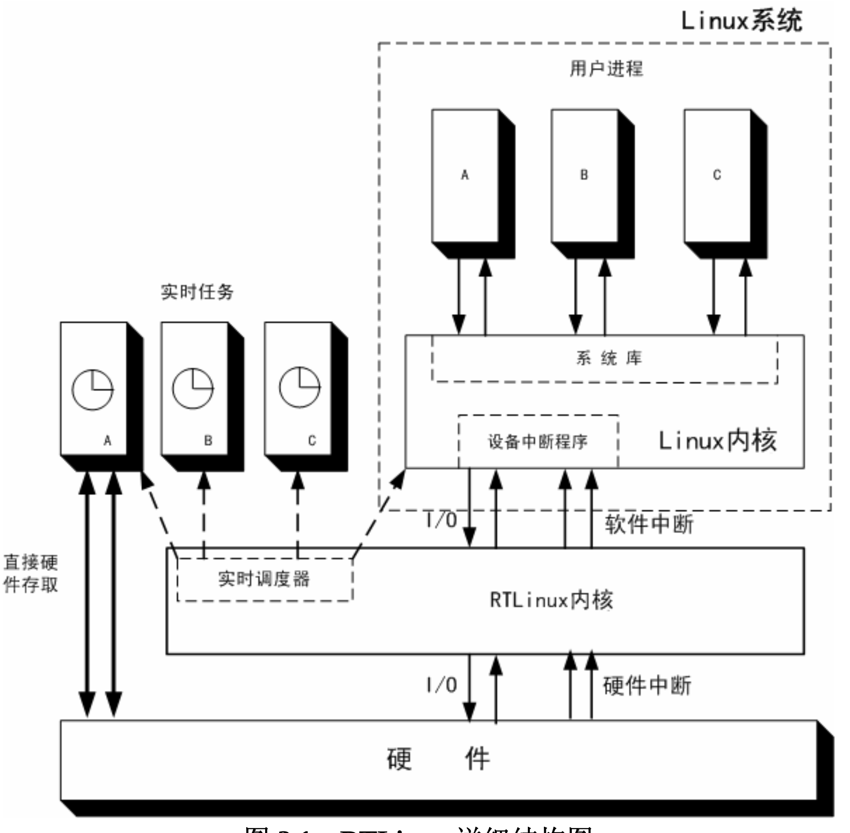


图 3.1 **RTLinux 详细结构图**

3.2 中断模拟

要在标准 Linux 上增加硬实时能力，首先遇到的一个问题是 Linux 为了达到同步使用关中断的方式。混杂在一块的关和开中断操作（i486 处理器的 cli 和 sti 机器指令）造成不可确定的中断分派延迟。Linux 内核是一整块大的内核。在提供系统服务各个部分之间没有一个保护的分界线。要改写 Linux 内核感到非常棘手。造成要限定关中断的时间非常困难，当更新版本发行时，也可能变得不正确。即使我们能处理这些，时间上离我们的要求仍然太长。

在实时 Linux 中，是通过在 Linux 内核与中断控制硬件之间增加一个模拟软件，这是与文献[6]相似的技术，但是用于不同的目的。在 Linux 源代码中所有的 cli，sti，和 iret（iret：中断返回指令）被替换为相应的宏：S_CLI，S_STI 和 S_IRET。所有的硬件中断指令都被中断模拟器捕捉。

```
/*  These are macros  */

S_CLI:   movl    $0, SFIF

S_IRET:  push    %ds
         pushl   %eax
         pushl   %edx
         movl    $KERNEL_DS, %edx
         mov     %dx, %ds
         cli
         movl    SFREQ, %edx
         andl    SFMASK, %edx
         bsrl    %edx, %eax
         jz      not_found
         movl    $0, SFIF
         sti
         jmp     SFIDT ( , %eax, 4)
not_fount:
         movl    $1, SFIF
         sti
         popl    %edx
         popl    %eax
         pop     %ds
         iret

S_STI:   pushfl
         pushl   $KERNEL_CS
         pushl   $done_STI
         S_IRET
done_STI:
```

程序 3.1 “软”cli，sti 和 iret

关中断发生时，在模拟器的一个变量重新设定。只要中断发生，模拟器将检查这个变量。假如这个值是已经设定（Linux 中断是允许的），Linux 的中断处理程序将立即调用。如果 Linux 中断是不允许的，中断处理程序将不会被调用。一个变量值将设定，并且保存所有挂起的中断的信息。一旦 Linux 中断允许处理时，所有挂起的中断将被处理。这种中断称之为软中断。

由于 Linux 不能直接控制中断控制器，Linux 的中断不会影响实时中断的处理。

S_CLI、S_STI 和 S_IRET 宏如程序 3.1 所示。这个代码使用 GNU 汇编规范。S_CLI 宏简单重新设定变量值，保存 Linux 中断状态。S_STI 宏设置正在被处理的中断的栈。S_IRET 宏模拟中断返回。S_IRET 宏的工作

就像硬件 iret 指令所做的那样允许软中断。

S_IRET 宏是三个宏中最有意思的一个。它先保存一些寄存器和初始化指向内核的数据段寄存器。然后存取全局变量。扫描所有挂起的中断而设置的屏蔽位。如果没有发现挂起的中断，设置中断状态变量，一个硬件的中断返回指令被执行。如果发现一个中断，跳转到 Linux 中断处理程序。中断处理程序返回后，依次跳转到下一个未处理中断的中断处理程序，直到没有中断再挂起为止。

扫描和转到中断处理程序是一个原子操作，否则，在这过程中有一个中断发生扫描将不能发现任何挂起的中断，这个新到的中断的处理程序将会延迟处理，直到下一个 S_STI 或者 S_IRET 被执行是才能被处理。

使用链式跳转的方式来代替子程序调用的方式调用 Linux 中断处理程序，是因为后者不能完全模拟直接的中断处理。Linux 中断处理程序检查栈来发现是用户还是内核代码被中断，基于这个做出决定处理。因此，保护中断状态是很重要的。

3.3 实时任务

实时任务是一个用户定义的程序，它按照在内核控制下的特定的调度方式来执行。

最开始的设计是给每一个实时任务有自己的地址空间来提供内存保护。这通过 80x86 处理器内置的分页机制[10]。在每次上下文切换中，页目录是基于寄存器的变化来指向新任务的页目录。

任务间的切换非常频繁，如果在 TLB 没有命中时，使得系统在上下文切换的开销很大，系统性能会降低。别的系统开销还有是系统的调用，在保护模式下也是个费时的操作。

一种提高性能的方法是所有的实时任务运行在一个地址空间。通过使用内核地址空间，除去了保护模式变换的系统开销。Linux 一个很有用的特性是：可装载内核模块。内核模块可以动态连接到内核地址空间，和链接为内核代码。每个模块定义了两个例程：init_module() 和 cleanup_module()。init_module()在模块装载到内核是调用，cleanup_module()在删除模块时调用。这就提供了一个简单的方法在 Linux 中操作驱动程序和文件系统。

可链接模块用以在当前的 RTLinux 中动态创建实时任务。这种实现方

法也更脆弱：一个实时任务的错误可能引起整个系统的崩溃。C 语言的使用加重了这个问题。数组、指针等的应用，很容易引起与内存相关的程序错误。另一方面，由于实时任务一般控制昂贵的外围设备，理所当然要使用与系统内核编程时相同的警告级别。

实时任务运行在内核地址空间有几个好处。除了上面提到的 TLB 命中问题和保护模式切换的问题外，这种方法使我们通过名字引用函数和对象，胜于通过描述符来引用。比如，实时任务表现为一个 C 的结构体。每个任务可以赋予一个 C 标识符，别的任务也可以通过这个标识符引用任务。动态链接执行过程中，模块装载解决了符号寻址问题，所以访问是非常高效的。

所有的任务在系统的地址空间，任务的切换也更简单。一个上下文切换是保存所有整数寄存器到栈中，改变栈的指针指向新的任务。同样也支持有浮点运算的任务。

为实时任务进行实时编程的接口将在第四章介绍。

3.3.1　实时线程数据结构

struct rtl_thread_struct

```
struct rtl_thread_struct {
     int *stack;
     int fpu_initialized;
     RTL_FPU_CONTEXT fpu_regs;
     int uses_fp;
     int *kmalloc_stack_bottom;
     struct rtl_sched_param sched_param; /* 线程调度参数 */
     struct rtl_thread_struct *next; /* 链表中下一个线程 */
     int cpu; /* 线程的 CPU 号 */
     hrtime_t resume_time; /* 恢复时间 */
     hrtime_t period; /* 任务周期 */
     hrtime_t timeval;
     struct module *creator; /* 线程创建者 */
     void (*abort)(void *);
     void *abortdata;
     int threadflags;
     rtl_sigset_t pending;
     rtl_sigset_t blocked;
     void *user[4];
     int errno_val;
     struct rtl_cleanup_struct *cleanup;
     int magic;
     struct rtl_posix_thread_struct posix_data;
```

```
    void *tsd [RTL_PTHREAD_KEYS_MAX];
};
```

程序 3.2 rtl_thread_struct 结构

3.3.2 创建线程和线程调度

一个实时程序使用一个或几个线程来执行。线程是轻量级进程，它们共享公共的地址空间。在 RTLinux 中，所有的线程共享 Linux 内核地址空间。

线程操作相关的函数将在第四章介绍。

3.4 实时调度

实时调度器的只要任务是满足所有实时任务的时间要求。有很多方法表示时间的约束和很多的调度策略[4]。不存在一个适合所有任务的调度策略。

在大多数实时系统中，调度器是由大的、复杂的代码块组成，它也不可能扩展到适用任何情况。用户只是通过调节参数来改变调度器的行为，往往这是不够的。一般调度器代码也比较慢。

在 RTLinux 中，允许用户编写自己的调度器代码。可以把它实现为一个可装载的内核模块。这就使得可以实验不同的调度策略和算法，以找到一个最适合自己应用的调度方式。

3.4.1 实现的调度器

迄今为止实现了两种调度器。一个是基于优先级的占先式调度器。调度策略如下所述。每个任务赋予一个唯一的优先级。假如有几个任务处于就绪状态，优先级最高的那个将运行。只要一个优先级更高的任务就绪，它就可以中断当前较低优先级任务的执行。每个任务假定它可以自由的放弃 CPU。

这个调度器直接支持周期任务。每个任务的周期和开始时间是可以给定的。一个中断驱动的（非周期的）任务通过定义中断处理程序，然后通过中断处理程序来唤醒相应的任务。

根据每个任务的周期和它们的终止时间，很自然的我们可以根据速率单调调度算法(rate monotonic scheduling algorithm, RMS)[5]决定每个任务

的优先级。根据这个算法，周期短的任务有高的优先级。对于有 n 个任务的实时调度来说，满足下面公式的实时任务将能够成功的调度，每个任务都不会超过它们的最终期限(deadline)：

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\cdots+\frac{C_n}{T_n}\le n(2^{1/n}-1)$$

这里，C_i 为任务 i 在每个周期的最长执行时间，T_i 为任务 i 的周期。非周期任务将处理为周期任务，同样赋予一个优先级[5]。

调度器把 Linux 当作一个有最低优先级的任务。Linux 只在没有实时任务运行时运行。为此，从 Linux 切换到实时任务时，软中断状态将被记录，而且禁止软中断。当切换回来时，软中断状态将恢复。

另外一个调度器是根据最早期限优先算法(Earliest Deadline First, EDF)实现的。在这个算法中没有静态的优先级。而是最靠近最终期限(deadline)的任务总是最先执行。

3.4.2 设计用户自己的调度器

RTLinux 作为开放的系统，具有以下的优势方便用户设计自己的调度器，以实现自己特有的调度方式：

- 一个分时复用的基于优先级调度的内核；
- 有精确可靠的时间片划分；
- 精确的时钟控制原语；
- 快且可预测的中断响应和进程切换时间；

RTLinux 的调度器在文件 rtl_sched.c 和 rtl_schedule.h 中定义，实时线程可以在创建时用函数 pthread_attr_setschedparam 设置或在运行中 pthread_setchedparam 改变其优先级。scheduler 将系统的优先级设为-1，而所有实时线程的优先级大于 0，从而保证实时线程优先执行。用户需要改动的是 rtl_schedule（调度过程）函数；两个重要的数据结构：schedule_t 和 rtl_thread_struct；和任务队列 task_queue *list，task_queue *destlist。

通过对以上的数据结构和调度过程的修改，用户可以自己实现特定的调度算法。

3.5 计时

精确的计时是正确的调度器操作是必须的。调度器常常要求在一个特定的时刻进行任务切换。计时的错误将引起背离计划的调度，导致任务释放抖动(task release jitter)[24]。在大多数的应用中任务释放抖动是不好的。要尽量减少它的影响。

低的时间精度一个原因是在操作系统中，使用周期的时钟中断。系统设计者必须在时钟中断处理函数开销与计时精度之间做一个折中[24]。有时候使用周期时钟并不能得到要求的计时器定时精度。

在 Linux 中也是一样。在 IBM 兼容的 PC 上，硬件定时器的时钟中断速率设定为大约 100Hz 左右。因此，任务可以达到 10 毫秒的精度。一些商业的操作系统，比如 VxWorks、REAL/IX 也是使用周期的时钟中断，尽管它们允许用户改变中断频率。

在 RTLinux 中，消除这个折中是在需要的时候才通过使用一个可编程间隔定时器来中断 CPU。特别地，使 Intel 8354 定时器芯片工作在 interrupt-on-terminal-count 模式。使用这种模式，可以使中断调度得到 1 微秒左右的精度。这种方法的定时器精度高而系统开销是最小。

为 Linux 模拟了一个周期的中断。使用软中断是非常简单的：为了模拟一个中断要求，一个未处理的中断屏蔽位被设定。在下一个软中断返回时，或者软 sti 执行时，处理函数将被调用。

3.5.1 时间相关函数

时间相关的函数将在第四章介绍。

3.6 进程间通信

由于 Linux 内核任何时候都可能被实时任务占先，Linux 线程不能安全地被实时任务调用。不管怎样，必须要有一些进程间通信(IPC)的机制。RTLinux 提供了三种通信方法。

3.6.1 FIFO 设备

RTLinux 用 FIFO 管道来在 Linux 进程或者 Linux 内核与实时进程间传递数据，这种管道称为实时管道(real-time FIFOs)，以区别 UNIX IPC 机制中的管道。

RT-FIFO 管道是在内核地址空间的。通过一个整数来引用。RT-FIFOs 的数目在编译系统时给定，可以重新编译系统改变大小。

操作 RT-FIFOs 的函数包括创建、删除、读 FIFO 和写 FIFO。读写操作是原子操作，不能中断。不可中断是为了避免优先级倒置问题。

在 Linux 进程中，把 RT-FIFOs 当作普通的字符设备，而不是一个系统调用。字符设备给用户一个与实时任务通信的全功能的应用程序接口(API)。这个接口对 Linux 进程来说是标准的设备接口，包括：open, close, read 和 write。

struct file_operations rtf_fops

```
static struct file_operations rtf_fops =
{
    rtf_llseek,
    rtf_read,
    rtf_write,
    NULL,
    rtf_poll,
    NULL,
    NULL,
    rtf_open,
    NULL,
    rtf_release,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};
```

下面是原始的存取例程：

```
int rtf_create(unsigned int minor, int size)
int rtf_destroy(unsigned int minor)
int rtf_put(unsigned int minor, void *buf, int count)
int rtf_get(unsigned int minor, void *buf, int count)
```

对于每个 FIFO 管道可以通过

```
int rtf_create_handler(unsigned int minor,
```

```
                    int (*handler) (unsigned int fifo))
```
安装自己的处理程序，当数据从 FIFO 中读出或写出时运行。

3.6.2 共享内存

在 RTLinux 启动的时候，通过指定内核一个 mem 参数决定内核可以使用的内存大小，空出来的内存空间用于实时任务和 Linux 进程进行通信的共享内存。在 RTLinux 任务中通过/dev/mem 设备在这段内存中寻址，Linux 进程也通过读取这段内存的数据获得实时任务提供的信息，这样完成实时任务和 Linux 进程之间的通信。

3.6.3 mbuff 驱动程序

它是由 Tomasz Motylewski 提供的一个使用共享内存的驱动程序，用来实现核心内存空间和用户之间的共享。通过使用 mbuff 提供的 mbuff_alloc()的函数给申请的内存取一个名字，mbuff 驱动程序使用一个链表通过这个名字来管理这些申请的内存。通过这个驱动程序也可以在包括 RTLinux 的 Linux 内核内存空间和用户内存空间之间共享内存。

```
#include <mbuff.h>
void * mbuff_alloc(const char *name, int size);
void mbuff_free(const char *name, void * mbuf);
```

第一次调用 mbuff_alloc 时，给定一个名字，一个给定大小的共享内存块将分配。这个内存块的引用数设为 1。调用成功返回新内存块的指针。失败时返回 NULL。如果给定的名字已经存在，将返回存在的内存块的指针，以操作这块共享内存块，该内存块的引用数将加 1。

第四章　RTLinux 应用程序设计

4.1　程序结构

每个实时应用程序可以分为两部分：实时部分和非实时部分[2]。非实时部分在用户空间执行，称为用户部分。实时部分要尽可能简单，只包含直接与时间相关的代码；由于硬件对时间的约束，低级的与硬件通信的代码一般也包含在实时部分。用户部分的代码主要实现为数据的处理，包括数据的发布、保存和用户界面。两部分之间的通信采用数据缓冲区。

图 4.1 所示的数据流程图是依照这个程序模型的典型实时应用程序。

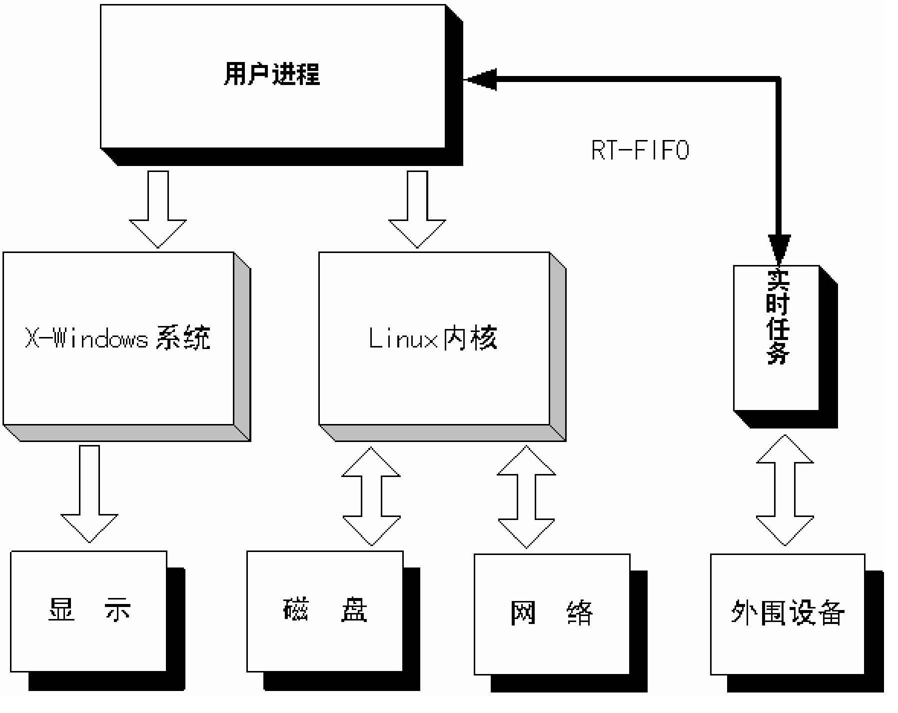


图 4.1　程序结构图

4.2 基本 API

4.2.1 POSIX 线程创建函数

就像前面介绍的那样，一个实时程序是由几个执行的线程组成的。线程是轻量级进程，它们共享共有的地址空间。在 RTLinux 中，所有的线程共享 Linux 内核地址空间。

int pthread_create (pthread_t *thread, pthread_attr_t * attr, void * (* start_routine)(void *), void *arg)

这是 RTLinux 的标准 POSIX 线程创建函数。这个线程运行函数指针 start_routine 指向的过程，arg 是这个函数的指针的入口参数。线程的属性由 attr 对象决定，可以为这个属性设置 CPU 号、堆栈大小等属性。设定若为 NULL，将会使用默认属性。返回 0 表示成功创建线程，线程号放在 thread 所指向的空间；返回非 0 表示创建失败。线程的属性决定在特定的 CPU 上创建线程(pthread_attr_setcpu_np)，是否使用 FPU(pthread_attr_setfp_np)。

int pthread_attr_init (pthread_attr_t *attr)

初始化线程运行的属性。

int pthread_ attr_setschedparam (pthread_attr_t *attr, const struct sched_param *param)和 int pthread_ attr_setschedparam (const pthread_attr_t *attr, struct sched_param *param)

这两个函数根据程序的需要相应地从 attr 中设定/取得线程的运行参数。param 是为调度的 SCHED_FIFO 和 SCHED_RR 策略定义的属性。

int pthread_attr_setcpu_np (pthread_atte_t *attr, int cpu)和
int pthread_attr_getcpu_np (pthread_atte_t *attr, int cpu)

设定/取得线程运行的 CPU 号。在 SMP 机器上允许线程在一个特定的 CPU 上运行。

int pthread_cancel (pthread_t thread)

取消一个运行的线程。

int pthread_delete_np (pthread_t thread)

删除一个线程，并且释放该线程的所有资源。返回 0 表示成功删除，非 0 表示删除失败。

pthrad_t pthread_self (void)

获得当前正在运行的线程号。

clockid_t rtl_getschedclock (void)

获得当前调度方法的时钟。

int rtl_setclockmode (clockid_t clock, int mode, hrtime_t mode_param)

设置当前的时钟模式，mode=RTL_CLOCK_MODE_ONESHOT 时是非周期（一次性）模式 mode_param 参数无用；mode=RTL_CLOCK_MODE_PERIODIC 时是周期模式，mode_param 参数是周期的长度。（有关时钟模式见 3.4 节的说明）

int pthread_wait_np (void)

当前周期的线程运行结束，总是返回 0。

4.2.2 时间相关函数

RTLinux 提供了一些时钟函数用于计时功能，包括线程调度，获得 TSP(timestamps)等。

下面的是一般的计时函数：

```
/* #include <rtl_time.h> */

int clock_gettime(clockid_t clock_id, struct timespec *ts);
hrtime_t clock_gethrtime(clockid_t clock);

struct timespec {
time_t tv_sec; /* 秒 */
long tv_nsec; /* 纳秒 */
};
```

clock_gettime：读取当前的时间，保存到 clock_id 所指的对象中。

clock_gethrtime：读取当前时间，但返回一个 64 位(hrtime_t)的纳秒时间值。

一些时间转换的函数，用于把时间格式转换为另外一种格式。

时间转换函数：

```
/*  #include <rtl_time.h>  */

hrtime_t timespec_to_ns(const struct timespec *ts); /*  timespec 到纳秒数转换  */
struct timespec timespec_from_ns(hrtime_t t); /*  纳秒数到 timespec 转换  */
const struct timespec * hrt2ts(hrtime_t value); /*
```

下面是一些支持的时钟类型。

时钟类型相关的宏：

- CLOCK_MONOTONIC: POSIX 时钟，以恒定速率运行；不会复位和调整
- CLOCK_REALTIME: 标准 POSIX 实时时钟。目前与 CLOCK_MONOTONIC 时钟相同
- CLOCK_RTL_SCHED: 调度器用来任务调度的时钟

以下是机器结构相关的时钟：

- CLOCK_8254: 在 x86 单处理器机器上用于调度的时钟
- CLOCK_APIC: 用在 SMP x86 机器的时钟

4.2.3 线程调度函数

RTLinux 提供一些调度方式，允许线程代码在特定的时刻运行。RTLinux 使用单纯优先级驱动的调度器，更搞优先级的线程总是被选择运行。如果两个线程的优先级拥有一样的优先级，选择那一个线程运行是不确定的。RTLinux 使用下面的调度 API：

int pthread_setschedparam (pthread_t thread, int policy, const struct sched_param *param)

设置一个线程的调度参数，用 policy 和 sched_param 两个参数设置 thread 的调度参数属性：

policy=SCHED_RR：使用 Round-Robin 方法调度

policy=SCHED_FIFO：使用先进先出的方法调度

返回 0 表示成功调度，非 0 表示失败。

int pthread_getschedparam (pthread_t thread, int policy, const struct sched_param *param)

获得一个线程的调度参数。将获得的 policy 和 sched_param 结构放在入口参数所指向的地址里面。

int pthread_make_periodic_np (pthread_t thread, hrtime start_time, hrtime_t period)

这个函数标记 thread 线程为可运行。线程将在 start_time 时刻开始运行，运行的时间间隔由 period 给定。

int pthread_wait_np (void)

pthread_wait_np 函数将挂起当前运行发线程直到下一周期。这个线程必须是 pthread_make_periodic_np 函数标记为可执行。

int sched_get_priority_max (int policy)和
int sched_get_priority_min (int policy)

确定 sched_priority 可能的值。

4.3 编程示例

前面介绍了 RTLinux 的基本 API，在这里以一个实例来说明 RTLinux 下的编程方法。这是一以测试 RTLinux 下中断延迟的程序。正如前面所说的，程序分为两部分，实时部分和非实时部分。实时部分通过使用一个模块，在将实时模块插入后，运行实时任务。对于非实时部分，实现对 FIFO 设备的读取，完成和实时任务的通信。



图 **4.2** 实时程序结构图

4.3.1 实时部分

init_module 完成对实时部分的初始化。cleanup_module 实现关闭实时模块的任务。

```
/*  * RTLinux scheduling accuracy measuring example  */

#include <rtl.h>
#include <rtl_fifo.h>
#include <time.h>
#include <rtl_sched.h>
#include <rtl_sync.h>
#include <pthread.h>
#include <unistd.h>
#include <rtl_debug.h>
#include <errno.h>
#include "common.h"

int ntests=500;
int period=1000000;
int bperiod=3100000;
int mode=0;
int absolute=0;
int fifo_size=4000;
int advance=0;

MODULE_PARM(period,"i");
MODULE_PARM(bperiod,"i");
MODULE_PARM(ntests,"i");
MODULE_PARM(mode,"i");
MODULE_PARM(absolute,"i");
MODULE_PARM(advance,"i");

pthread_t thread;
int fd_fifo;


void *thread_code(void *param)
{
 hrtime_t expected;
 hrtime_t diff;
 hrtime_t now;
 hrtime_t last_time = 0;
 hrtime_t min_diff;
 hrtime_t max_diff;
 struct sample samp;
 int i;
 int cnt = 0;
```

```
 int cpu_id = rtl_getcpuid();

 rtl_printf ("Measurement task starts on CPU %d\n", cpu_id);
 if (mode) {
     int ret = rtl_setclockmode (CLOCK_REALTIME,
RTL_CLOCK_MODE_PERIODIC, period);

     if (ret != 0) {
         conpr("Setting periodic mode failed\n");
         mode = 0;
     }
 } else {

     rtl_setclockmode (CLOCK_REALTIME, RTL_CLOCK_MODE_ONESHOT,
0);
 }

 expected = clock_gethrtime(CLOCK_REALTIME) + 2 * (hrtime_t) period;


 fd_fifo = open("/dev/rtf0", O_NONBLOCK);
 if (fd_fifo < 0) {
     rtl_printf("/dev/rtf0 open returned %d\n", fd_fifo);
     return (void *) -1;
 }

 if (advance) {
     rtl_stop_interrupts(); /* Be careful with this! The task won't
be preempted by anything else. This is probably only appropriate
for small high-priority tasks. */
 }

 /* first cycle */
 clock_nanosleep (CLOCK_REALTIME, TIMER_ABSTIME, hrt2ts(expected
- advance), NULL);
 expected += period;
 now = clock_gethrtime(CLOCK_MONOTONIC);
 last_time = now;

 do {
     min_diff = 2000000000;
     max_diff = -2000000000;

     for (i = 0; i < ntests; i++) {
         ++cnt;
         clock_nanosleep (CLOCK_REALTIME, TIMER_ABSTIME,
hrt2ts(expected - advance), NULL);

         now = clock_gethrtime(CLOCK_MONOTONIC);
         if (absolute && advance && !mode) {
             if (now < expected) {
```

```
                rtl_delay (expected - now);
            }
            now = clock_gethrtime(CLOCK_MONOTONIC);
        }
        if (absolute) {
            diff = now - expected;
        } else {
            diff = now - last_time - period;
            if (diff < 0) {
                diff = -diff;
            }
        }
        if (diff < min_diff) {
            min_diff = diff;
        }
        if (diff > max_diff) {
            max_diff = diff;
        }

        expected += period;
        last_time = now;
    }

    samp.min = min_diff;
    samp.max = max_diff;
    write (fd_fifo, &samp, sizeof(samp));
 } while (1);
 return 0;
}

pthread_t background_threadid;

void *background_thread(void *param)
{
 hrtime_t next = clock_gethrtime(CLOCK_REALTIME);
 while (1) {
     hrtime_t t = gethrtime ();
     next += bperiod;
     /* the measurement task should preempt the following loop */
     while (gethrtime() < t + bperiod * 2 / 3);
    clock_nanosleep (CLOCK_REALTIME, TIMER_ABSTIME, hrt2ts(next),
NULL);
 }
}


int init_module(void)
{
 pthread_attr_t attr;
 struct sched_param sched_param;
 int thread_status;
```

```
      int fifo_status;

 rtf_destroy(0);
 fifo_status = rtf_create(0, fifo_size);
 if (fifo_status) {
      rtl_printf("RTLinux measurement test fail.
fifo_status=%d\n",fifo_status);
      return -1;
 }


 rtl_printf("RTLinux measurement module on CPU
%d\n",rtl_getcpuid());
 pthread_attr_init (&attr);
 if (rtl_cpu_exists(1)) {
      pthread_attr_setcpu_np(&attr, 1);
 }
 sched_param.sched_priority = 1;
 pthread_attr_setschedparam (&attr, &sched_param);
 rtl_printf("About to thread create\n");
 thread_status = pthread_create (&thread,  &attr, thread_code,
(void *)1);
 if (thread_status != 0) {
      rtl_printf("failed to create RT-thread: %d\n",
thread_status);
      return -1;
 } else {
      rtl_printf("created RT-thread\n");
 }

 if (bperiod) {
      pthread_create (&background_threadid,  NULL,
background_thread, NULL);
 }
 return 0;
}


void cleanup_module(void)
{
      rtl_printf ("Removing module on CPU %d\n", rtl_getcpuid());
 pthread_cancel (thread);
 pthread_join (thread, NULL);
 close(fd_fifo);
 rtf_destroy(0);
 if (bperiod) {
      pthread_cancel (background_threadid);
      pthread_join (background_threadid, NULL);
 }
}
```

4.3.2 非实时部分

非实时部分实际上建行就是一个应用程序，可以在控制台下编写。不过必须有/dev/rtf0 设备的读取权限才能运行。

```
#include <stdio.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <rtl_fifo.h>
#include "common.h"



int main()
{
 int fd0;
 int n;
 struct sample samp;

 if ((fd0 = open("/dev/rtf0", O_RDONLY)) < 0) {
     fprintf(stderr, "Error opening /dev/rtf0\n");
     exit(1);
 }


 while (1) {
     n = read(fd0, &samp, sizeof(samp));
     printf("min: %8d, max: %8d\n", (int) samp.min, (int)
samp.max);
     fflush(stdout);
 }

 return 0;
}
```

4.3.3 编译和运行程序

在一台 Celeron 412MHz，196MB 内存，RTLinux3.1 的机器上进行如下的 Makefile 编译：

```
all: rt_process.o irqsema.o monitor histplot

include ../../rtl.mk

monitor: monitor.c
```

```
 $(CC) ${USER_CFLAGS} ${INCLUDE} -Wall -O2 -o monitor monitor.c

clean:
 rm -f *.o monitor histplot periodic_monitor gnuplot.out

include $(RTL_DIR)/Rules.make
```

则程序可以测试调度的时间精度，程序的运行结果为：

```
min:      0, max:   24480
min:      0, max:   10720
min:      0, max:   10912
min:      0, max:   10976
min:      0, max:   11072
min:      0, max:   10656
min:      0, max:   10944
min:      0, max:   11200
min:      0, max:   11200
min:      0, max:   11008
min:      0, max:   10912
.........
```

第五章　RTLINUX 的性能测试

根据 2.2 节有关中断延迟的介绍。在这一章，我们将对 RTLinux 的中断延迟进行测试。测试的机器为 Celeron 412MHz，196MB 内存，RTLinux3.1+Linux-2.2.19（与前面测试的 Linux 版本相同）的机器上进行了测试（测试程序在附录 2）。测试的结果如表 5.1 所示：

负载类型	平均值	最小值	最大值
无负载	2.36	2.10	15.50
硬盘循环拷贝	3.20	2.10	19.40
计算负载	2.56	2.20	14.50

表 5.1　实时中断延迟时间（单位：微秒）

- 无负载：所有进程已经杀死
- 硬盘循环拷贝：一个硬盘循环拷贝 shell 脚本在运行
- 计算负载：一个循环执行浮点运算的 C 程序

从测试结果看，RTLinux 的中断响应时间明显小于标准 Linux 的中断响应时间。在磁盘拷贝负载下最大延迟为 19.40 微秒，在这种负载情况下，标准 Linux 的延迟达到了 500 微秒。

为了计算调度精度，我们运行了一个周期实时任务。在每个周期任务唤醒时，记录下并比较它的时间，记录下最大的时间值。时间值在 10 微秒左右。

从上结果看，RTLinux 是完全可以胜任实时运算的操作系统。

第六章　感想与体会

当前，以信息家电为中心的嵌入式系统正得到蓬勃的发展。并且现在出现的市场只是冰山一角。我相信，随着嵌入式系统和网络技术的发展，整个世界将更加网络化，计算化。

世界是丰富多彩的，因而无处不在的嵌入式系统也要适应这个千变万化的世界。嵌入式系统也是千变万化的！嵌入式系统的这种特点注定了其市场的碎片化。任何公司都没有足够的能量统一市场。这带来新的游戏规则，给新生的公司以机会。

开放源码软件的特点注定了它非常适合嵌入式系统。开放源码导致定制的方便，我们完全可以从多个软件中提取需要的精华，应用到自己的应用上去，大大节省了开发成本。我认为以服务为主导，以开放源码软件为支柱的公司将是嵌入式系统市场上的重要角色。

我们知道，形成规模才能导致成本的降低，而在多样化的嵌入式系统中，更重要的是定制，是千变万化的应用环境。所以，如果利用封闭源码的商业软件，由于不能形成规模成本将非常高。当然，也不排除在某些领域像 Windows CE, PalmOS 等成熟的系统形成优势，但从长远的角度来看，开放源码软件更加适合嵌入式系统。我们期待着更多成熟的开放源码软件的出现，并且认为商品化是开放源码软件很好的出路。也希望中国的软件业能在开放源码软件革命中获得一席之地。

几乎以免费方式就可以获得的 Linux，其商业价值到底在哪里？换句话说，到底什么样的系统最需要 Linux？我认为下面三种系统最应该使用 Linux：

- 安全相关的系统。这些系统应该使用开放源码的系统，以防止封闭系统留下的各种后门。这包括关键的网络服务器、政府部门和军用服务器、工作站等等。
- 实时系统。对各种实时性系统来讲，底层的操作系统同样不能建立在黑箱之上。这包括工业控制用的实时系统，各种军用系统等。
- 嵌入式系统。嵌入式系统是高度定制的系统，对嵌入式系统而言，没有也不应该有通用的系统。为了达到定制，就需要开放操作系统的源码。这也是原先许多 Windows CE 开发商现在转向 Linux 的原因。

因此，实时嵌入式系统正是 Linux 大显身手的地方。国内外的许多开发商早就看到了这一点，并在嵌入式 Linux 的开发方面投入了大量的人力和资金。综合说来，以 Linux 作为操作系统开发嵌入式系统，有如下优势：

- Linux 的可移植性好，开发工具丰富。每个 CPU 开发商在发布每一款新的嵌入式 CPU 时，均会投入大量人力移植 Linux 内核，并提供丰富的开发工具。
- 开放源码。Linux 内核的源码开放，可大大方便系统的定制开发。
- 可用资源丰富。在 Linux 系统上进行开发时，可以获得的源代码等资源比其他封闭系统多得多。
- 成本低。Linux 的低成本特点，可大大降低最终嵌入式系统的成本，并扩大开发商的获利空间。

参考文献

[1] *吴朝晖教授纵谈嵌入式技术*. 微电脑世界. 2000 年第 49 期

[2] 邹思铁. *嵌入式 Linux 设计与应用*. 清华大学出版社. 2002.1

[3] 刘云新，张尧学. *一个基于 Linux 的嵌入式实时操作系统*. 计算机工程与应用. 2001.7: 64-85.

[4] D. M. Dhamdhere. *Systems Programming and Operating Systems*. 电子工业出版社. 2001.9

[5] William Stallings. *Operating Systems Internals and Design Principles*. 电子工业出版社. 2001.6

[6] H. Lyckama & L. Bayer. *UNIX time-sharing system: The MERT Operating System*. Bell System Technical Journal. 57(6):2049-2086, 1978.

[7] Alessandro Rubini. *Linux 驱动程序*. 中国电力出版社. 2000.4

[8] *嵌入式论文集*. 电子产品世界. 2000.11

[9] 沈绪榜，何立民主编. *2001 嵌入式系统及单片机国际学术交流论文集*. 北京航空航天大学出版社. 2001.10

[10] Daniel P.Bovet & Marco Cesati. *深入理解 Linux 内核*. 中国电力出版社. 2001.10

[11] Andrew S. Tanenbaum & Albert S. Woodhull. *Operating Systems Design and Implementation*. 电子工业出版社. 2001.4

[12] Scott Maxwell. *Linux 内核源代码分析*. 机械工业出版社. 2000.6

[13] Jean J.Labrosse. *uC/OS the Real-Time Kernel. R&D Publications*,1992

[14] Michael Barabanov. *A Linux-based Real-Time Operating System[D]*. Master Paper. New Mexico Institute of Technology, 1997.6

[15] Victor Yodaiken, Michael Barabanov. *A Real-Time Linux*. New Mexico Institute of Technology

[16] Maurice J.Bach. *The Design of the UNIX Operating sytem*. 机械工业出版社. 2000.4

[17] Dennis W.Ritchie & Ken Thempson. *The UNIX time-sharing System*. Communications of the Association for computing Machinery, 17(1):365-375, July 1974.

[18] Gabriel A.Wainer. *Implementing real-time Service in MINX*. Operating Systems Review, 29(3):75-84, July 1995.

[19] http://www.microsoft.com/

[20] http://www.palmos.com/
[21] http://www.epoccity.com/
[22] http://www.linux.org/
[23] http://www.xlinux.com.cn/
[24] http://www.rtlinux.org/
[25] http:// www.realtime-info.be/
[26] http://www.qnx.com
[27] http://www.vxworks.com
[28] http://www.aero.polimi.it
[29] http://www.ittc.ukans.edu/kurt
[30] http://www.minigui.org/
[31] http://www.gnu.org/
[32] http://www.linuxdoc.org/
[33] http://www.linuxaid.com.cn

致 谢

我要感谢 Linus Torvalds，是他发明了 Linux，并把它贡献出来。使得我可以接触最先进的编程思想，使得我可以进入到操作系统里面，了解系统的行为。还有 Victor Yodaiken 和 Michael Barabanov，是他们提出了在 Linux 上建立实时内核的方法。并且开发了 RTLinux。

也要感谢朱利华、张建业、唐高友、陈娜、雷娟等几位同学。创新实验室是一个温暖、和谐的集体，能在这样一个实验室，我觉得很幸运。

我出身幸福的家庭，在我的整个求学生涯中，我的父母付出了无数的汗水，他们的鼓励、关怀，他们为我所做的一切，都不能用语言来表达。

附录 A

中断延迟测试代码：

这是一个补丁(patch)文件，在我的 Linux-2.2.12 版本下编译通过。通过下面的命令粘贴到 Linux 源文件中：

```
# cd /usr/src/linux-2.2.12
# patch –p1 < /usr/src/linux-2.2.12/interrupt-latency-2.2.12-patch
```

粘贴完成后编译 Linux 内核：

```
# cd /usr/src/linux-2.2.12
# make config        # 配置内核
# make dep
# make bzImage       # 编译内核
# make modules       # 编译模块
# make modules_install      # 装载模块
# cp arch/i386/boot/bzImage /boot/interrupt-lattency
```

配置 LILO，编辑/etc/lilo.conf，把下面几行添加进：

```
image=/boot/interrupt-latency
        label=interrupt-lattency
        read-only
        root=/dev/hda1
```

重新装 LILO：

```
# /sbin/lilo
```

重启后在 LILO 提示符下输入：

```
LILO: interrupt-latency
```

则可以监控系统调用开/关中断的内核就起动完成，还需要运行一个用户空间程序(view.c，见下面)用于输入命令和打印测试结果。

interrupt-latency-2.2.12-patch 文件如下：

```
diff --exclude=version.h --exclude=config.h -Nru linux-2.2.12/arch/i386/kernel/Makefile
linux-2.2.12-interrupt/arch/i386/kernel/Makefile
--- linux-2.2.12/arch/i386/kernel/Makefile    Wed Jan 20 10:18:53 1999
+++ linux-2.2.12-interrupt/arch/i386/kernel/Makefile   Mon Mar  6 11:04:25 2000
@@ -14,7 +14,9 @@

 O_TARGET := kernel.o
 O_OBJS   := process.o signal.o entry.o traps.o irq.o vm86.o \
-         ptrace.o ioport.o ldt.o setup.o time.o sys_i386.o
+         ptrace.o ioport.o ldt.o setup.o time.o sys_i386.o \
+        intr_blocking.o
+
 OX_OBJS  := i386_ksyms.o
 MX_OBJS  :=

diff --exclude=version.h --exclude=config.h -Nru linux-2.2.12/arch/i386/kernel/entry.S
linux-2.2.12-interrupt/arch/i386/kernel/entry.S
--- linux-2.2.12/arch/i386/kernel/entry.S     Fri Apr 30 08:13:37 1999
+++ linux-2.2.12-interrupt/arch/i386/kernel/entry.S    Mon Mar  6 11:05:10 2000
@@ -96,6 +96,7 @@
      movl %dx,%es;

 #define RESTORE_ALL   \
+     incl  intrData; \
      popl %ebx;     \
      popl %ecx;     \
      popl %edx;     \
@@ -562,6 +563,8 @@
      .long SYMBOL_NAME(sys_ni_syscall)           /* streams1 */
      .long SYMBOL_NAME(sys_ni_syscall)           /* streams2 */
      .long SYMBOL_NAME(sys_vfork)         /* 190 */
+
+     .long SYMBOL_NAME(sys_get_intrData)    /* 191 */

      /*
       * NOTE!! This doesn't have to be exact - we just have
diff --exclude=version.h --exclude=config.h -Nru linux-2.2.12/arch/i386/kernel/head.S
linux-2.2.12-interrupt/arch/i386/kernel/head.S
--- linux-2.2.12/arch/i386/kernel/head.S      Thu Jan 14 22:57:25 1999
+++ linux-2.2.12-interrupt/arch/i386/kernel/head.S     Mon Mar  6 11:14:26 2000
@@ -316,6 +316,7 @@
      movl %ax,%es
      pushl $int_msg
```

```
	call SYMBOL_NAME(printk)
+	incl  intrData
	popl %eax
	popl %ds
	popl %es
diff --exclude=version.h --exclude=config.h -Nru
linux-2.2.12/arch/i386/kernel/intr_blocking.c
linux-2.2.12-interrupt/arch/i386/kernel/intr_blocking.c
--- linux-2.2.12/arch/i386/kernel/intr_blocking.c	Wed Dec 31 16:00:00 1969
+++ linux-2.2.12-interrupt/arch/i386/kernel/intr_blocking.c	Mon Mar  6 11:41:50 2000
@@ -0,0 +1,322 @@
+#include <asm/system.h>
+
+
+/**** platform ****/
+#define readclock(low) \
+    __asm__ __volatile__ ("rdtsc" : "=a" (low) : : "edx")
+
+/**** configure ****/
+#define	NUM_LOG_ENTRY	4
+#define	INTR_IENABLE	0x200
+
+/**** data structure ****/
+struct IntrData {
+   /* count interrupt and iret */
+   int breakCount;
+
+   /* the test name */
+   const char * testName;
+
+   /* flag to control logging */
+   unsigned logFlag;  /* 0 - no logging; 1 - logging */
+
+   /* panic flag - set to 1 if something is realy wrong */
+   unsigned panicFlag;
+
+   /* for synchro between start and end */
+   unsigned syncFlag;
+
+   /* we only log interrupts within certain range */
+   unsigned rangeLow;
+   unsigned rangeHigh;
+
+   /* count the total number interrupts  and intrs in range*/
+   unsigned numIntrs;
+   unsigned numInRangeIntrs;
+
+
+   /* error accounting */
+   unsigned skipSti;
+   unsigned skipCli;
```

```
+    unsigned syncStiError;
+    unsigned syncCliError;
+    unsigned stiBreakError;
+    unsigned restoreSti;
+    unsigned restoreCli;
+
+    struct {
+       /* worst blocking time */
+       unsigned blockingTime;
+
+       const char * startFileName;
+       unsigned startFileLine;
+       unsigned startCount;
+
+       const char *endFileName;
+       unsigned endFileLine;
+       unsigned endCount;
+    } count[NUM_LOG_ENTRY];
+};
+
+struct IntrData intrData = {
+    0,
+    "interrupt latency test (4 distinctive entries)",
+    0,
+    0,
+    0,
+
+    1,
+    0xffffffff,
+
+    0,
+    0,
+
+    0,
+    0,
+    0,
+    0,
+    0,
+    0,
+    0
+};
+
+
+/**** functions ****/
+#if 0
+void intr_check_int(int x)
+{
+
+    unsigned flag;
+    __intr_save_flags(flag);
+
```

```
+   if ((flag & INTR_IENABLE) != 0) {
+  switch(x) {
+   case 0 :
+      intrData.count[0].blockingTime ++;
+      break;
+   case 1 :
+      intrData.count[0].startFileLine ++;
+      break;
+   case 2 :
+      intrData.count[0].startCount ++;
+      break;
+   case 3 :
+      intrData.count[0].endFileLine ++;
+      break;
+   case 4 :
+      intrData.count[0].endCount ++;
+      break;
+   default :
+      intrData.count[0].startFileName = "Wrong check number";
+      break;
+  }
+  } else {
+  switch(x) {
+   case 0 :
+      intrData.count[1].blockingTime ++;
+      break;
+   case 1 :
+      intrData.count[1].startFileLine ++;
+      break;
+   case 2 :
+      intrData.count[1].startCount ++;
+      break;
+   case 3 :
+      intrData.count[1].endFileLine ++;
+      break;
+   case 4 :
+      intrData.count[1].endCount ++;
+      break;
+   default :
+      intrData.count[1].startFileName = "Wrong check number";
+      break;
+  }
+  }
+}
+
+#endif
+
+
+static inline void intr_SetPanic(unsigned x, const char *fname, unsigned l)
+{
+   if (intrData.panicFlag != 0) {
```

```
+       /* double error; impossible */
+       intrData.panicFlag = 99;
+       return;
+    }
+    intrData.panicFlag = x;
+    intrData.count[0].startFileName = fname;
+    intrData.count[0].startFileLine = l;
+}
+
+static const char *intrStartFileName;
+static unsigned intrStartFileLine;
+static unsigned intrStartCount;
+
+/* strategy :
+ * if it is true "cli", i.e., clearing the IF, we remember
+ * everything, and clear breakCount.
+ */
+void intr_cli(const char *fname, unsigned lineno)
+{
+    unsigned flag;
+    __intr_save_flags(flag);
+
+    __intr_cli();
+
+    /* if we are not logging or we have an error, do nothing */
+    if ((intrData.logFlag == 0) || ( intrData.panicFlag != 0)) {
+       return;
+    }
+
+    /* do nothing we had IF cleared before we call this function */
+    if ((flag & INTR_IENABLE) == 0) {
+       intrData.skipCli ++;
+       return;
+    }
+
+    /* debug */
+    if (intrData.syncFlag == 1) {
+       intrData.syncCliError ++;
+    }
+    intrData.syncFlag = 1;
+
+    intrData.breakCount = 0;
+
+    /* Read the Time Stamp Counter */
+    intrStartFileName = fname;
+    intrStartFileLine = lineno;
+    readclock(intrStartCount);
+}
+
+
+/* strategy:
```

```
+ * we do a count only if
+ * 1. syncFlag is 1 (a valid cli() was called)
+ * 2. breakCount is 0 (no iret is called between cli() and this sti()
+ */
+void intr_sti(const char *fname, unsigned lineno)
+{
+   unsigned flag;
+   unsigned endCount;
+   unsigned diff;
+   int i;
+
+   __intr_save_flags(flag);
+
+   /* if we are not logging or we have an error, do nothing */
+   if ((intrData.logFlag == 0) || ( intrData.panicFlag != 0)) {
+      __intr_sti();
+      return;
+   }
+
+
+   /* check if this is a real sti() */
+   if ((flag & INTR_IENABLE) != 0) {
+      intrData.skipSti ++;
+      __intr_sti();
+      return;
+   }
+
+   /* check 1*/
+   if (intrData.syncFlag != 1) {
+      intrData.syncStiError ++;
+      __intr_sti();
+      return;
+   }
+
+   /* check 2 */
+   if (intrData.breakCount != 0) {
+      intrData.stiBreakError ++;
+      __intr_sti();
+      return;
+   }
+
+   /* read count again */
+   readclock(endCount);
+
+   intrData.syncFlag = 0;
+
+   diff = endCount - intrStartCount;
+
+   if ((diff >= intrData.rangeLow) && (diff <= intrData.rangeHigh)) {
+      unsigned lowest=0xffffffff;
+      unsigned lowestIndex;
```

```
+       unsigned sameIndex = 0xffffffff;
+
+       intrData.numInRangeIntrs++;
+
+       /* check if we need to log this event */
+       for (i=0; i< NUM_LOG_ENTRY; i++) {
+
+          if (lowest > intrData.count[i].blockingTime) {
+             lowest = intrData.count[i].blockingTime;
+              lowestIndex = i;
+          }
+
+          if ( (lineno == intrData.count[i].endFileLine) &&
+               (intrStartFileLine == intrData.count[i].startFileLine) &&
+              (fname[0] == intrData.count[i].endFileName[0]) &&
+               (intrStartFileName[0] == intrData.count[i].startFileName[0]) ) {
+                /* if the line numbers are same, the first chars in
+                 * both file names are same, we consider it is the same
+                 * entry. */
+                sameIndex = i;
+          }
+       }
+
+       if (sameIndex == 0xffffffff)  {
+          i = lowestIndex;
+       } else {
+          i = sameIndex;
+       }
+
+       if (diff > intrData.count[i].blockingTime) {
+          intrData.count[i].blockingTime = diff;
+          intrData.count[i].endFileName = fname;
+          intrData.count[i].endFileLine = lineno;
+          intrData.count[i].endCount = endCount;
+          intrData.count[i].startFileName = intrStartFileName;
+          intrData.count[i].startFileLine = intrStartFileLine;
+          intrData.count[i].startCount = intrStartCount;
+       }
+    }
+
+    intrData.numIntrs++;
+    __intr_sti();
+}
+
+
+void intr_restore_flags(const char *fname, unsigned lineno, unsigned x)
+{
+    unsigned flag;
+
+    /* if we are not logging or we have an error, do nothing */
+    if ((intrData.logFlag == 0) || ( intrData.panicFlag != 0)) {
```

```
+       __intr_restore_flags(x);
+       return;
+   }
+
+   __intr_save_flags(flag);
+
+   if (((flag & INTR_IENABLE) == 0)  &&
+       ((x & INTR_IENABLE) != 0) )  {
+       intrData.restoreSti ++;
+       intr_sti(fname, lineno);
+   }
+
+   if ( ((flag & INTR_IENABLE) != 0) &&
+        ((x & INTR_IENABLE) == 0) ) {
+       intrData.restoreCli ++;
+       intr_cli(fname, lineno);
+   }
+
+   __intr_restore_flags(x);
+}
+
+#include <asm/uaccess.h>
+
+asmlinkage int sys_get_intrData(void ** ptr)
+{
+    return put_user(&intrData, ptr);
+}
diff --exclude=version.h --exclude=config.h -Nru linux-2.2.12/include/asm-i386/system.h
linux-2.2.12-interrupt/include/asm-i386/system.h
--- linux-2.2.12/include/asm-i386/system.h     Mon Oct 11 21:28:12 1999
+++ linux-2.2.12-interrupt/include/asm-i386/system.h   Mon Mar  6 11:08:02 2000
@@ -174,13 +174,33 @@
 #define wmb() __asm__ __volatile__ ("": : :"memory")

 /* interrupt control.. */
+#if 0
 #define __sti() __asm__ __volatile__ ("sti": : :"memory")
 #define __cli() __asm__ __volatile__ ("cli": : :"memory")
 #define __save_flags(x) \
 __asm__ __volatile__("pushfl ; popl %0":"=g" (x): /* no input */ :"memory")
 #define __restore_flags(x) \
 __asm__ __volatile__("pushl %0 ; popfl": /* no output */ :"g" (x):"memory")
+#endif

+#define __intr_sti() __asm__ __volatile__ ("sti": : :"memory")
+#define __intr_cli() __asm__ __volatile__ ("cli": : :"memory")
+#define __intr_save_flags(x) \
+__asm__ __volatile__("pushfl ; popl %0":"=g" (x): /* no input */ :"memory")
+#define __intr_restore_flags(x) \
+__asm__ __volatile__("pushl %0 ; popfl": /* no output */ :"g" (x):"memory")
+
```

```
+/* jsun */
+extern void intr_cli(const char *, unsigned);
+extern void intr_sti(const char *, unsigned);
+extern void intr_restore_flags(const char *, unsigned, unsigned);
+extern void intr_sync_flag(const char *, unsigned lineno);
+
+#define __cli() intr_cli(__FILE__, __LINE__)
+#define __sti() intr_sti(__FILE__, __LINE__)
+#define __save_flags(x) \
+__asm__ __volatile__("pushfl ; popl %0":"=g" (x): /* no input */ :"memory")
+#define __restore_flags(x) intr_restore_flags(__FILE__, __LINE__, x)

 #ifdef __SMP__

@@ -197,9 +217,8 @@

 #define cli() __cli()
 #define sti() __sti()
-#define save_flags(x) __save_flags(x)
+#define save_flags(x) __save_flags(x)
 #define restore_flags(x) __restore_flags(x)
-
 #endif

 /*
```

用户空间程序 **view.c** 如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <assert.h>

/****************** CONFIG ****************/
#define       NUM_LOG_ENTRY        4
#define     CLOCK_FREQUENCY      266 /* cycles per microsecond */
#define     SYSCALL_NUMBER       191

/****************** END OF CONFIG ***************/

#define       CMD_EXIT            0
#define       CMD_DISPLAY          1
#define       CMD_START           2
#define       CMD_STOP            3
#define       CMD_CONTINUE         4
#define       CMD_SET_RANGE        5
#define       CMD_LAST               6

struct IntrData {
```

```
   /* count interrupt and iret */
   int breakCount;

   /* the test name */
   const char * testName;

   /* flag to control logging */
   unsigned logFlag;  /* 0 - no logging; 1 - logging */

   /* panic flag - set to 1 if something is realy wrong */
   unsigned panicFlag;

   /* for synchro between start and end */
   unsigned syncFlag;

   /* we only log interrupts within certain range */
   unsigned rangeLow;
   unsigned rangeHigh;

   /* count the total number interrupts  and intrs in range*/
   unsigned numIntrs;
   unsigned numInRangeIntrs;


   /* error accounting */
   unsigned skipSti;
   unsigned skipCli;
   unsigned syncStiError;
   unsigned syncCliError;
   unsigned stiBreakError;
   unsigned restoreSti;
   unsigned restoreCli;

   struct {
      /* worst blocking time */
      unsigned blockingTime;

      const char * startFileName;
      unsigned startFileLine;
      unsigned startCount;

      const char *endFileName;
      unsigned endFileLine;
      unsigned endCount;
   } count[NUM_LOG_ENTRY];
};



unsigned pData;
struct IntrData data;
```

```
int kmem;
char buf[81];

unsigned int GetInt(const char *prompt)
{
   unsigned int i;
   printf("%s", prompt);
   scanf("%d", &i);
   return i;
}

unsigned int GetHex(const char *prompt)
{
   unsigned int i;
   printf("%s", prompt);
   scanf("%x", &i);
   return i;
}

unsigned GetCommand()
{
   for(;;) {
      unsigned cmd;
      printf("\n");
      printf("Command Menu \n");
      printf("============\n");
      printf("0-Exit  1-Display 2-Start logging 3-Stop logging 4-Continue\n");
      printf("5-Set range\n");
      printf("\n");
      printf("Your choice : ");
      scanf("%u", &cmd);
      if ((cmd >= 0) && (cmd < CMD_LAST)) {
         return cmd;
      } else {
         printf("Invalid choice!!!!\n\n");
      }
   }
}

unsigned GetKmemInt(unsigned offset)
{
   off_t seekError;
   ssize_t size;
   unsigned num=0;

   assert((offset & 3) == 0);

   seekError = lseek(kmem, offset, SEEK_SET);
   assert(seekError != (off_t)-1);

   size = read(kmem, &num, sizeof(num));
```

```
    assert(sizeof(num) == 4);
    assert(size == 4);
    return num;
}

char * GetKmemString(unsigned offset)
{
    off_t seekError;
    ssize_t size;

    buf[80]=0;
    buf[0]=0;

    if (offset == 0) return buf;

    seekError = lseek(kmem, offset, SEEK_SET);
    assert(seekError != (off_t)-1);

    size = read(kmem, buf, 80);
    assert(size == 80);
    return buf;
}

void GetKmemBlock(unsigned offset, void * buf, unsigned bufSize)
{
    off_t seekError;
    ssize_t size;

    seekError = lseek(kmem, offset, SEEK_SET);
    assert(seekError != (off_t)-1);

    size = read(kmem, buf, bufSize);
    assert(size == bufSize);
}

void SetKmemBlock(unsigned offset, void *buf, unsigned bufSize)
{
    off_t seekError;
    ssize_t size;

    seekError = lseek(kmem, offset, SEEK_SET);
    assert(seekError != (off_t)-1);

    size = write(kmem, buf, bufSize);
    assert(size == bufSize);
}

void SetKmemInt(unsigned offset, unsigned x)
{
    off_t seekError;
    ssize_t size;
```

```
   seekError = lseek(kmem, offset, SEEK_SET);
   assert(seekError != (off_t)-1);

   size = write(kmem, &x, sizeof(x));
   assert(size == sizeof(x));
}


void Display()
{
   unsigned i;

   GetKmemBlock(pData, &data, sizeof(data));
   printf("%s : \n", GetKmemString((unsigned)data.testName));
   printf("breakCount    : %d\n", data.breakCount);
   printf("logFlag      : %d\n", data.logFlag);
   printf("panicFlag    : %d\n", data.panicFlag);
   printf("syncFlag     : %d\n", data.syncFlag);

   printf("range        : [%u(0x%x) : %u(0x%x)]\n",
        data.rangeLow, data.rangeLow,
        data.rangeHigh, data.rangeHigh);

   printf("numIntrs     : %u\n", data.numIntrs);
   printf("numInRangeInts: %u\n", data.numInRangeIntrs);

   printf("skipSti skipCli syncSti syncCli stiBreak restSti restCli\n");
   printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t\n",
        data.skipSti,
        data.skipCli,
        data.syncStiError,
        data.syncCliError,
        data.stiBreakError,
        data.restoreSti,
        data.restoreCli);


   for (i=0; i< NUM_LOG_ENTRY; i++) {
      printf("log entry : %d\n", i);
      printf("\tblockingTime    : %u (%u us)\n",
        data.count[i].blockingTime,
        data.count[i].blockingTime / CLOCK_FREQUENCY);
      printf("\tstartFileName   : %s\n",
           GetKmemString((unsigned)data.count[i].startFileName));
      printf("\tstartFileLine   : %u\n", data.count[i].startFileLine);
      printf("\tstartCount      : %u\n", data.count[i].startCount);
      printf("\tendFileName     : %s\n",
           GetKmemString((unsigned)data.count[i].endFileName));
      printf("\tendFileLine     : %u\n", data.count[i].endFileLine);
      printf("\tendCount        : %u\n", data.count[i].endCount);
```

```
    }
    printf("\n");
}

void StartLogging()
{
    unsigned i;

    /* init first */
    data.breakCount = 0;
    data.logFlag = 0;
    data.panicFlag = 0;
    data.syncFlag = 0;

    data.numIntrs = 0;
    data.numInRangeIntrs = 0;

    data.skipSti =
        data.skipCli =
        data.syncStiError =
        data.syncCliError =
        data.stiBreakError =
        data.restoreSti =
        data.restoreCli = 0;

    for (i=0; i< NUM_LOG_ENTRY; i++) {
        data.count[i].blockingTime = 0;
        data.count[i].startFileName = 0;
        data.count[i].startFileLine = 0;
        data.count[i].startCount = 0;
        data.count[i].endFileName = 0;
        data.count[i].endFileLine = 0;
        data.count[i].endCount = 0;
    }

    SetKmemBlock(pData, &data, sizeof(data));

    /* turn the logging flag */
    SetKmemInt(pData + ((unsigned)&data.logFlag - (unsigned)&data), 1);
}

void EndLogging()
{
    /* turn the logging flag */
    SetKmemInt(pData + ((unsigned)&data.logFlag - (unsigned)&data), 0);
}

void ContinueLoggin()
{
    /* turn the logging flag */
    SetKmemInt(pData + ((unsigned)&data.logFlag - (unsigned)&data), 1);
```

```
}

void SetRange()
{
   data.rangeLow = GetInt("Input lower bound (decimal) : ");
   printf("\tlower bound is %u(0x%x)\n", data.rangeLow, data.rangeLow);

   data.rangeHigh = GetInt("Input upper bound (decimal) : ");
   printf("\tupper bound is %u(0x%x)\n", data.rangeHigh, data.rangeHigh);

   SetKmemInt(pData + ((unsigned)&data.rangeLow - (unsigned)&data),
           data.rangeLow);
   SetKmemInt(pData + ((unsigned)&data.rangeHigh - (unsigned)&data),
           data.rangeHigh);
}


main()
{
   unsigned int cmd;
   unsigned long offset;

   kmem = open("/dev/kmem", O_RDWR);
   assert(kmem > 0);

   if (syscall(SYSCALL_NUMBER, &pData) != 0) {
      printf("failed to get jsunData address through syscall 191!\n");
      pData = GetHex("Input manually the address of jsunData : ");
   }

   GetKmemBlock(pData, &data, sizeof(data));

   for(;;) {
      cmd = GetCommand();

      switch (cmd) {
       case CMD_DISPLAY:
         Display();
         break;

       case CMD_START:
         StartLogging();
         break;

       case CMD_STOP:
         EndLogging();
         break;

       case CMD_CONTINUE:
         ContinueLoggin();
         break;
```

```
      case CMD_SET_RANGE:
        SetRange();
        break;

      case CMD_EXIT:
        close(kmem);
        return;

      default:
        assert(0 == 1);
      }
   }
}
```

附录 B

上下文切换测试程序 lat_ctx.c：

```
/*
 * lat_ctx.c - context switch timer
 *
 * usage: lat_ctx [-s size] #procs [#procs....]
 *
 * Copyright (c) 1994 Larry McVoy.  Distributed under the FSF GPL with
 * additional restriction that results may published only if
 * (1) the benchmark is unmodified, and
 * (2) the version in the sccsid below is included in the report.
 * Support for this development by Sun Microsystems is gratefully
acknowledged.
 */
char *id = "$Id$\n";

#include "bench.h"

#if defined(sgi) && defined(PIN)
#include <sys/sysmp.h>
#include <sys/syssgi.h>
int ncpus;
#endif

#define MAXPROC 2048
#define CHUNK   (4<<10)
#define TRIPS   5
#ifndef max
#define max(a, b)    ((a) > (b) ? (a) : (b))
#endif

int process_size, *data; /* size & pointer to an array that big */
int pids[MAXPROC];
int p[MAXPROC][2];
double  pipe_cost(int p[][2], int procs);
int ctx(int procs, int nprocs);
int sumit(int);
void killem(int procs);
void doit(int p[MAXPROC][2], int rd, int wr);
int create_pipes(int p[][2], int procs);
int create_daemons(int p[][2], int pids[], int procs);

int
main(int ac, char **av)
{
    int i, max_procs;
    double  overhead = 0;
```

```
    if (ac < 2) {
usage:      printf("Usage: %s [-s kbytes] processes
[processes ...]\n",
            av[0]);
        exit(1);
    }

    /*
     * Need 4 byte ints.
     */
    if (sizeof(int) != 4) {
        fprintf(stderr, "Fix sumit() in ctx.c.\n");
        exit(1);
    }

    /*
     * If they specified a context size, get it.
     */
    if (!strcmp(av[1], "-s")) {
        if (ac < 4) {
            goto usage;
        }
        process_size = atoi(av[2]) * 1024;
        if (process_size > 0) {
            data = (int *)calloc(1, max(process_size, CHUNK));
            BENCHO(sumit(CHUNK), sumit(0), 0);
            overhead = gettime();
            overhead /= get_n();
            overhead *= process_size;
            overhead /= CHUNK;
        }
        ac -= 2;
        av += 2;
    }

#if defined(sgi) && defined(PIN)
    ncpus = sysmp(MP_NPROCS);
    sysmp(MP_MUSTRUN, 0);
#endif
    for (max_procs = atoi(av[1]), i = 1; i < ac; ++i) {
        int procs = atoi(av[i]);
        if (max_procs < procs) max_procs = procs;
    }
    max_procs = create_pipes(p, max_procs);
    overhead += pipe_cost(p, max_procs);
    max_procs = create_daemons(p, pids, max_procs);
    fprintf(stderr, "\n\"size=%dk ovr=%.2f\n", process_size/1024,
overhead);
    for (i = 1; i < ac; ++i) {
        double  time;
        int procs = atoi(av[i]);
```

```
        if (procs > max_procs) continue;

        BENCH(ctx(procs, max_procs), 0);
        time = usecs_spent();
        time /= get_n();
        time /= procs;
        time /= TRIPS;
        time -= overhead;
        fprintf(stderr, "%d %.2f\n", procs, time);
    }

    /*
     * Close the pipes and kill the children.
     */
    killem(max_procs);
    for (i = 0; i < max_procs; ++i) {
        close(p[i][0]);
        close(p[i][1]);
        if (i > 0) {
            wait(0);
        }
    }
    return (0);
}


int
ctx(int procs, int nprocs)
{
    int msg;
    int i;
    int sum;

    /*
     * Main process - all others should be ready to roll, time the
     * loop.
     */
    for (i = 0; i < TRIPS; ++i) {
        if (write(p[nprocs - procs][1], &msg, sizeof(msg)) !=
           sizeof(msg)) {
            perror("read/write on pipe");
            exit(1);
        }
        if (read(p[nprocs-1][0], &msg, sizeof(msg)) != sizeof(msg))
{
            perror("read/write on pipe");
            exit(1);
        }
        sum = sumit(process_size);
    }
```

```
    return (sum);
}

void
killem(int procs)
{
    int i;

    for (i = 1; i < procs; ++i) {
        if (pids[i] > 0) {
            kill(pids[i], SIGTERM);
        }
    }
}

void
doit(int p[][2], int rd, int wr)
{
    int msg, sum = 0 /* lint */;

    signal(SIGTERM, SIG_DFL);
    if (data) bzero((void*)data, process_size);
    for ( ;; ) {
        if (read(p[rd][0], &msg, sizeof(msg)) != sizeof(msg)) {
            perror("read/write on pipe");
            break;
        }
        sum = sumit(process_size);
        if (write(p[wr][1], &msg, sizeof(msg)) != sizeof(msg)) {
            perror("read/write on pipe");
            break;
        }
    }
    use_int(sum);
    exit(1);
}

int
doit_cost(int p[][2], int procs)
{
    static  int k;
    int msg = 1;
    int i;

    for (i = 0; i < TRIPS; ++i) {
        if (write(p[k][1], &msg, sizeof(msg)) != sizeof(msg)) {
            perror("read/write on pipe");
            exit(1);
        }
        if (read(p[k][0], &msg, sizeof(msg)) != sizeof(msg)) {
            perror("read/write on pipe");
```

```
            exit(1);
        }
        if (++k == procs) {
            k = 0;
        }
    }
    return (msg);
}

/*
 * The cost returned is the cost of going through one pipe once in usecs.
 * No memory costs are included here, this is different than lmbench1.
 */
double
pipe_cost(int p[][2], int procs)
{
    double  result;

    /*
     * Measure the overhead of passing a byte around the ring.
     */
    BENCH(doit_cost(p, procs), 0);
    result = usecs_spent();
    result /= get_n();
    result /= TRIPS;
    return result;
}

int
create_daemons(int p[][2], int pids[], int procs)
{
    int i;
    int msg;

    /*
     * Use the pipes as a ring, and fork off a bunch of processes
     * to pass the byte through their part of the ring.
     *
     * Do the sum in each process and get that time before moving on.
     */
    signal(SIGTERM, SIG_IGN);
    for (i = 1; i < procs; ++i) {
        switch (pids[i] = fork()) {
           case -1:  /* could not fork, out of processes? */
            procs = i;
            break;

           case 0:   /* child */
#if defined(sgi) && defined(PIN)
            sysmp(MP_MUSTRUN, i % ncpus);
#endif
```

```
            doit(p, i-1, i);
            /* NOTREACHED */

           default:  /* parent */
            ;
        }
    }

    /*
     * Go once around the loop to make sure that everyone is ready and
     * to get the token in the pipeline.
     */
    if (write(p[0][1], &msg, sizeof(msg)) != sizeof(msg) ||
       read(p[procs-1][0], &msg, sizeof(msg)) != sizeof(msg)) {
        perror("write/read/write on pipe");
        exit(1);
    }
    if (data) bzero((void*)data, process_size);
    return procs;
}

int
create_pipes(int p[][2], int procs)
{
    int i;
    /*
     * Get a bunch of pipes.
     */
    morefds();
    for (i = 0; i < procs; ++i) {
        if (pipe(p[i]) == -1) {
            return i;
        }
    }
    return procs;
}

/*
 * Bring howmuch data into the cache, assuming that the smallest cache
 * line is 16 bytes.
 */
int
sumit(int howmuch)
{
    int done, sum = 0;
    register int *d = data;

#if 0
#define A   sum+=d[0]+d[4]+d[8]+d[12]+d[16]+d[20]+d[24]+d[28]+\
        d[32]+d[36]+d[40]+d[44]+d[48]+d[52]+d[56]+d[60]+\
        d[64]+d[68]+d[72]+d[76]+d[80]+d[84]+d[88]+d[92]+\
```

```
        d[96]+d[100]+d[104]+d[108]+d[112]+d[116]+d[120]+d[124];\
        d+=128;
#define TWOKB   A A A A
#else
#define A
    sum+=d[0]+d[1]+d[2]+d[3]+d[4]+d[5]+d[6]+d[7]+d[8]+d[9]+\

    d[10]+d[11]+d[12]+d[13]+d[14]+d[15]+d[16]+d[17]+d[18]+d[19]+\

    d[20]+d[21]+d[22]+d[23]+d[24]+d[25]+d[26]+d[27]+d[28]+d[29]+\

    d[30]+d[31]+d[32]+d[33]+d[34]+d[35]+d[36]+d[37]+d[38]+d[39]+\

    d[40]+d[41]+d[42]+d[43]+d[44]+d[45]+d[46]+d[47]+d[48]+d[49]+\

    d[50]+d[51]+d[52]+d[53]+d[54]+d[55]+d[56]+d[57]+d[58]+d[59]+\

    d[60]+d[61]+d[62]+d[63]+d[64]+d[65]+d[66]+d[67]+d[68]+d[69]+\

    d[70]+d[71]+d[72]+d[73]+d[74]+d[75]+d[76]+d[77]+d[78]+d[79]+\

    d[80]+d[81]+d[82]+d[83]+d[84]+d[85]+d[86]+d[87]+d[88]+d[89]+\

    d[90]+d[91]+d[92]+d[93]+d[94]+d[95]+d[96]+d[97]+d[98]+d[99]+\
        d[100]+d[101]+d[102]+d[103]+d[104]+\
        d[105]+d[106]+d[107]+d[108]+d[109]+\
        d[110]+d[111]+d[112]+d[113]+d[114]+\
        d[115]+d[116]+d[117]+d[118]+d[119]+\
        d[120]+d[121]+d[122]+d[123]+d[124]+d[125]+d[126]+d[127];\
        d+=128;  /* ints; bytes == 512 */
#define TWOKB   A A A A
#endif

    for (done = 0; done < howmuch; done += 2048) {
        TWOKB
    }
    return (sum);
}


/*  bench.h  */

#ifndef _BENCH_H
#define _BENCH_H

#ifdef WIN32
#include <windows.h>
typedef unsigned char bool_t;
#endif

#include <assert.h>
#include      <ctype.h>
```

```
#include      <stdio.h>
#ifndef WIN32
#include      <unistd.h>
#endif
#include      <stdlib.h>
#include      <fcntl.h>
#include      <signal.h>
#include      <errno.h>
#ifndef WIN32
#include      <strings.h>
#endif
#include      <sys/types.h>
#ifndef WIN32
#include      <sys/mman.h>
#endif
#include      <sys/stat.h>
#ifndef WIN32
#include      <sys/wait.h>
#include <time.h>
#include      <sys/time.h>
#include      <sys/socket.h>
#include      <sys/un.h>
#include      <sys/resource.h>
#define PORTMAP
#include <rpc/rpc.h>
#endif

#ifndef HAVE_uint
typedef unsigned int uint;
#endif

#ifdef HAVE_uint64_t
typedef uint64_t uint64;
#else
typedef unsigned long long uint64;
#endif

#define NO_PORTMAPPER    /* needs to be up here, lib_*.h look at it
*/
#include "stats.h"
#include "timing.h"
#include "lib_tcp.h"
#include "lib_udp.h"
#include "lib_unix.h"


#ifdef  DEBUG
#   define      debug(x) fprintf x
#else
#   define      debug(x)
#endif
```

```
#ifdef  NO_PORTMAPPER
#define TCP_SELECT   -31233
#define TCP_XACT -31234
#define TCP_CONTROL  -31235
#define TCP_DATA -31236
#define TCP_CONNECT  -31237
#define UDP_XACT -31238
#define UDP_DATA -31239
#else
#define TCP_SELECT   (u_long)404038   /* XXX - unregistered */
#define TCP_XACT (u_long)404039   /* XXX - unregistered */
#define TCP_CONTROL  (u_long)404040   /* XXX - unregistered */
#define TCP_DATA (u_long)404041   /* XXX - unregistered */
#define TCP_CONNECT  (u_long)404042   /* XXX - unregistered */
#define UDP_XACT     (u_long)404032   /* XXX - unregistered */
#define UDP_DATA     (u_long)404033   /* XXX - unregistered */
#define VERS     (u_long)1
#endif

#define UNIX_CONTROL "/tmp/lmbench.ctl"
#define UNIX_DATA    "/tmp/lmbench.data"
#define UNIX_LAT "/tmp/lmbench.lat"

/*
 * socket send/recv buffer optimizations
 */
#define SOCKOPT_READ 0x0001
#define SOCKOPT_WRITE    0x0002
#define SOCKOPT_RDWR 0x0003
#define SOCKOPT_PID  0x0004
#define SOCKOPT_REUSE    0x0008
#define SOCKOPT_NONE 0

#ifndef SOCKBUF
#define SOCKBUF      (1024*1024)
#endif

#ifndef XFERSIZE
#define XFERSIZE (64*1024)    /* all bandwidth I/O should use this */
#endif

#if defined(SYS5) || defined(WIN32)
#define bzero(b, len)    memset(b, 0, len)
#define bcopy(s, d, l)   memcpy(d, s, l)
#define rindex(s, c) strrchr(s, c)
#endif
#define gettime      usecs_spent
#define streq        !strcmp
#define ulong        unsigned long

#ifdef USE_RAND
```

```
#define srand48      srand
#define drand48()    ((double)rand() / (double)RAND_MAX)
#endif
#ifdef USE_RANDOM
#define srand48      srand
#define drand48()    ((double)rand() / (double)RAND_MAX)
#endif

#ifdef WIN32
#include <process.h>
#define getpid _getpid
int gettimeofday(struct timeval *tv, struct timezone *tz);
#endif

#define  SMALLEST_LINE    32      /* smallest cache line size */
#define  TIME_OPEN2CLOSE

#define  GO_AWAY  signal(SIGALRM, exit); alarm(60 * 60);
#define  REAL_SHORT     50000
#define  SHORT         1000000
#define  MEDIUM        2000000
#define  LONGER        7500000 /* for networking data transfers */
#define  ENOUGH       REAL_SHORT

#define  TRIES        11

typedef struct {
    int N;
    uint64   u[TRIES];
    uint64   n[TRIES];
} result_t;
void   insertinit(result_t *r);
void   insertsort(uint64, uint64, result_t *);
void save_median();
void save_minimum();
void save_results(result_t *r);
void get_results(result_t *r);


#define  BENCH0(loop_body, overhead_body, enough) {             \
    int      __i, __N;                          \
    double       __oh;                          \
    result_t     __overhead, __r;               \
    insertinit(&__overhead); insertinit(&__r);        \
    __N = (enough == 0 || get_enough(enough) <= 100000) ? TRIES : 1;\
    if (enough < LONGER) {loop_body;} /* warm the cache */     \
    for (__i = 0; __i < __N; ++__i) {             \
        BENCH1(overhead_body, enough);                \
        if (gettime() > 0)                      \
            insertsort(gettime(), get_n(), &__overhead);  \
        BENCH1(loop_body, enough);                \
```

```
        if (gettime() > 0)                   \
            insertsort(gettime(), get_n(), &__r);     \
    }                               \
    for (__i = 0; __i < __r.N; ++__i) {              \
        __oh = __overhead.u[__i] / (double)__overhead.n[__i];  \
        __r.u[__i] -= (uint64)((double)__r.n[__i] * __oh); \
    }                               \
    save_results(&__r);                      \
}

#define BENCH(loop_body, enough) {                    \
    long     __i, __N;                  \
    result_t     __r;                   \
    insertinit(&__r);                        \
    __N = (enough == 0 || get_enough(enough) <= 100000) ? TRIES : 1;\
    if (enough < LONGER) {loop_body;} /* warm the cache */     \
    for (__i = 0; __i < __N; ++__i) {              \
        BENCH1(loop_body, enough);               \
        if (gettime() > 0)                   \
            insertsort(gettime(), get_n(), &__r);     \
    }                               \
    save_results(&__r);                      \
}

#define BENCH1(loop_body, enough) {                   \
    double       __usecs;               \
    BENCH_INNER(loop_body, enough);              \
    __usecs = gettime();                     \
    __usecs -= t_overhead() + get_n() * l_overhead();      \
    settime(__usecs >= 0. ? (uint64)__usecs : 0.);         \
}

#define BENCH_INNER(loop_body, enough) {              \
    static u_long    __iterations = 1;               \
    int      __enough = get_enough(enough);          \
    u_long       __n;                   \
    double       __result = 0.;              \
                                    \
    while(__result < 0.95 * __enough) {              \
        start(0);                       \
        for (__n = __iterations; __n > 0; __n--) {     \
            loop_body;                  \
        }                           \
        __result = stop(0,0);                    \
        if (__result < 0.99 * __enough               \
           || __result > 1.2 * __enough) {           \
            if (__result > 150.) {               \
                double   tmp = __iterations / __result;    \
                tmp *= 1.1 * __enough;               \
                __iterations = (u_long)(tmp + 1); \
            } else {                    \
```

```
                 if (__iterations > (u_long)1<<27) {   \
                     __result = 0.;            \
                     break;                \
                 }                     \
                 __iterations <<= 3;           \
             }                         \
         }                             \
    } /* while */                              \
    save_n((uint64)__iterations); settime((uint64)__result);   \
}


/*
 * Generated from msg.x which is included here:

    program XACT_PROG {
       version XACT_VERS {
        char
        RPC_XACT(char) = 1;
    } = 1;
    } = 3970;

 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <rpc/types.h>

#define XACT_PROG ((u_long)404040)
#define XACT_VERS ((u_long)1)
#define RPC_XACT ((u_long)1)
#define RPC_EXIT ((u_long)2)
extern char *rpc_xact_1();
extern char *client_rpc_xact_1();

#endif /* _BENCH_H */
```